VELOCTY : AN OPTIMIZING STATIC COMPILER FOR MATLAB AND PYTHON

by

Sameer Jagdale

School of Computer Science
McGill University, Montréal

A THESIS SUBMITTED TO THE FACULTY OF GRADUATE STUDIES AND RESEARCH
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE

Abstract

High-level scientific languages such as MATLAB and Python's NumPy library are gaining popularity among scientists and mathematicians. These languages provide many features such as dynamic typing, high-level scientific functions etc. which allow easy prototyping. However these features also inhibit performance of the code.

We present Velo**C**ty, an optimizing static compiler for MATLAB and Python as a solution to the problem of enhancing performance of programs written in these languages. In most programs, a large portion of the time is spent executing a small part of the code. Moreover, these sections can often be compiled ahead of time and improved performance can be achieved by optimizing only these 'hot' sections of the code. Velo**C**ty takes as input functions written in MATLAB and Python specified by the user and generates an equivalent C++ version. Velo**C**ty also generates glue code to interface with MATLAB and Python. The generated code can then be compiled and packaged as a shared library that can be linked to any program written in MATLAB and Python. We also implemented optimisations to eliminate array bounds checks, reuse previously allocated memory during array operations and support parallel execution using OpenMP.

Velo**C**ty uses the Velociraptor toolkit. We implemented a C++ backend for the Velociraptor intermediate representation, VRIR, and language-specific runtimes for MATLAB and Python. We have also implemented a MATLAB VRIR generator using the *Mc***LAB** toolkit.

Velo**C**ty was evaluated using 17 MATLAB benchmarks and 9 Python benchmarks. The MATLAB benchmark versions compiled using Velo**C**ty with all optimisations enabled were between 1.3 to 458 times faster than the MathWorks' MATLAB 2014b interpreter and JIT compiler. Similarly, Python benchmark versions were between 44.11 and 1681 times faster than the CPython interpreter.

Résumé

Les langages scientifiques de haut niveau, tels que MATLAB et Python et sa librairie NumPy, gagnent en popularité auprès des scientifiques et des mathématiciens. Ces langages offrent des fonctionalités telles que le typage dynamique et des fonctions scientifiques de haut niveau qui permettent un prototypage facile. Par contre, ces fonctionalités diminue la performance en exécution du code.

Nous présentons VeloCty, un compilateur statique optimisant pour MATLAB et Python comme solution au problème d'améliorer la performances des programmes écrits dans ces langages. Pour la majorité des programmes, une grande proportion du temps d'exécution est passée à exécuter une petite section du code. De plus, ces sections peuvent souvent être compilées avant l'exécution du code et on peut obtenir une amélioration en performance en optimisant seulement ces sections chaudes. VeloCty prend en entrée des fonctions écrites en MATLAB et Python spécifiées par l'utilisateur et génère une version équivalente en C++. VeloCty génère également le code d'interfaçage pour l'intégration avec MATLAB et Python. Le code généré peut ainsi être compilé comme une bibliothèque partagée qui peut être liée avec n'importe quel programme écrit en MATLAB et Python. Nous implémentons aussi des optimisations pour éliminer les tests de bornes des tableaux, pour réutiliser de la mémoire déjà allouée dans les opérations sur les tableaux, et pour supporter l'exécution parallèle via OpenMP.

VeloCty utilise le système de compilation Velociraptor. Nous implémentons un générateur de code qui transforme la représentation intermédiaire de Velociraptor, VRIR, en C++ ainsi que des supports d'exécution spécifiques pour MATLAB et Python. Nous avons également implémenté un générateur de code MATLAB à VRIR à l'aide de McLab.

VeloCty a été évalué avec des programmes de test de performance, 17 écrits en MAT-

LAB et 9 écrits en Python. Les résultats de VeloCty en utilisant toutes nos optimisations sur les tests en MATLAB montrent qu'il est 1.3 à 458 fois plus rapide que l'interpréteur et le compilateur en-ligne de MATLAB 2014b par MathWorks. Pour les tests en Python, VeloCty est 44.11 à 1681 fois plus rapide que l'interpréteur CPython.

Acknowledgements

I am thankful to my supervisor, Prof. Laurie Hendren, whose help and encouragement has made this thesis possible. It is because of her that I will graduate from the Master's program with a greater understanding of compilers as well as a greater respect for them .

I would also like to thank Rahul Garg, who developed the Velociraptor framework on which this research relies upon heavily. Moreover, I would like to thank him for suggesting this line of research and for mentoring me throughout the course of my research.

Additionally, I would like to thank Vineet Kumar, Ismail Badawi and Xu Li who helped me understand the *McLAB* framework as well as Erick Lavoie and Vincent Foley-Bourgon who helped me translate the abstract in French. I would also like to thank my other lab mates, Sujay Kathrotia, Faiz Khan, Andrew Bodzay and Lei Lopez who made working in the lab fun.

I would also like to thank my parents, my brother and all of my friends old and new, who never stopped supporting me and without whom I would not be where I am today.

Finally, I would like to thank the wonderful city of Montreal, whose beauty and people have made my Master's experience magical and memorable.

This work was supported, in part, by the Natural Sciences and Engineering Research Council of Canada (NSERC).

Table of Contents

List of Figures

List of Tables

List of Listings

Chapter 1
Introduction

With the advent of multicore processors, there has been a renewed interest in the development of performance tools and algorithms targeted for parallel architectures. Many research areas provide a wide variety of problems which would show improved performance when executed in parallel. One such area is scientific and numerical computing. Scientific algorithms are used by researchers from various fields such as chemistry, biology, geography etc. as well as different sub-fields of computer science like machine learning. In most cases, these algorithms are written in languages that are collectively known as array-based languages. A few examples of such languages are MATLAB [Matb], Julia[BKSE12] and Python[Foua] with it's NumPy[Dev] library.

Array-based languages offer features like, an interpreter style read-eval-print-loop, functions such as eval and feval for dynamic code evaluation, no types etc. which enable rapid prototyping. However due to the very same features, these languages show poorer performance when compared to statically compiled languages. A common approach for improving the performance is compile whole programs to languages such as FORTRAN[GNUa] and C[Rit]. However, in most cases, the most computationally intensive portion of the program is small, often localised inside a loop body. Hence compiling the entire program is not necessary. In most cases speed up observed through partial compilation of hot code sections is commensurate with that observed by compiling the whole program. This allows the user to continue programming in the language he/she is more comfortable in. Additionally, these functions may be resusable for other programs. In such cases, the functions will

have to be compiled only once and can be reused for the other programs.

This thesis addresses the problem of improving the performance of programs written in array-based languages by compiling the hot sections to parallel C++[Foud]. We support both MATLAB and NumPy. There are two main challenges. First one is supporting the different and often complementary semantics of both languages. The other is supporting the large number of builtins methods that are supported by both languages. Our solution implement a static C++ backend for Velociraptor[GH14] toolkit and use tools to compile MATLAB and Python programs to the Velociraptor intermediate representation, VRIR. The *Mc*LAB [CLD+10] static pipeline is used for MATLAB and PyVrir, a Python frontend for Velociraptor is used for Python.

1.1 VeloCty Compilation Pipeline

The compilation pipeline for Velo**C**ty can be seen in *Figure 1.1*. As mentioned earlier, PyVrir is a proof of concept Python frontend that is part of the Velociraptor framework and the MATLAB frontend is written using the *Mc*LAB frontend. In the *Mc*LAB pipeline a MATLAB program is parsed by the *Mc*LAB frontend and converted into an AST based representation known as McAST. The McSAF[Doh11] framework then performs various analyses such as kind analysis[DHR11] and function lookup on McAST and then generates another AST based representation called McLAST. The framework also performs a transformation from the colon expression to the range expression that was implemented as part of this thesis. Additional details on the transformation can be found in Section 3.8. McLAST is then converted to Tame IR[DH12] by the Tamer[DH12] framework. Tame IR is a three address representation of MATLAB. Analyses such as value analysis, shape analysis, isComplex analysis and IntegerOk analysis are performed on this IR. These analyses provide information on the type, dimensions and complexity of different variables code which is useful for generating the VRIR and subsequently the C++ code. The IntegerOk analysis identifies variables which can safely be declared as integers in the target language. This analysis is useful since MATLAB defines all variables as double by default. Tame IR is then given as input to Tamer+, a code aggregation framework, which generates the high-level McLAST representation from it. Code generated from McLAST is devoid

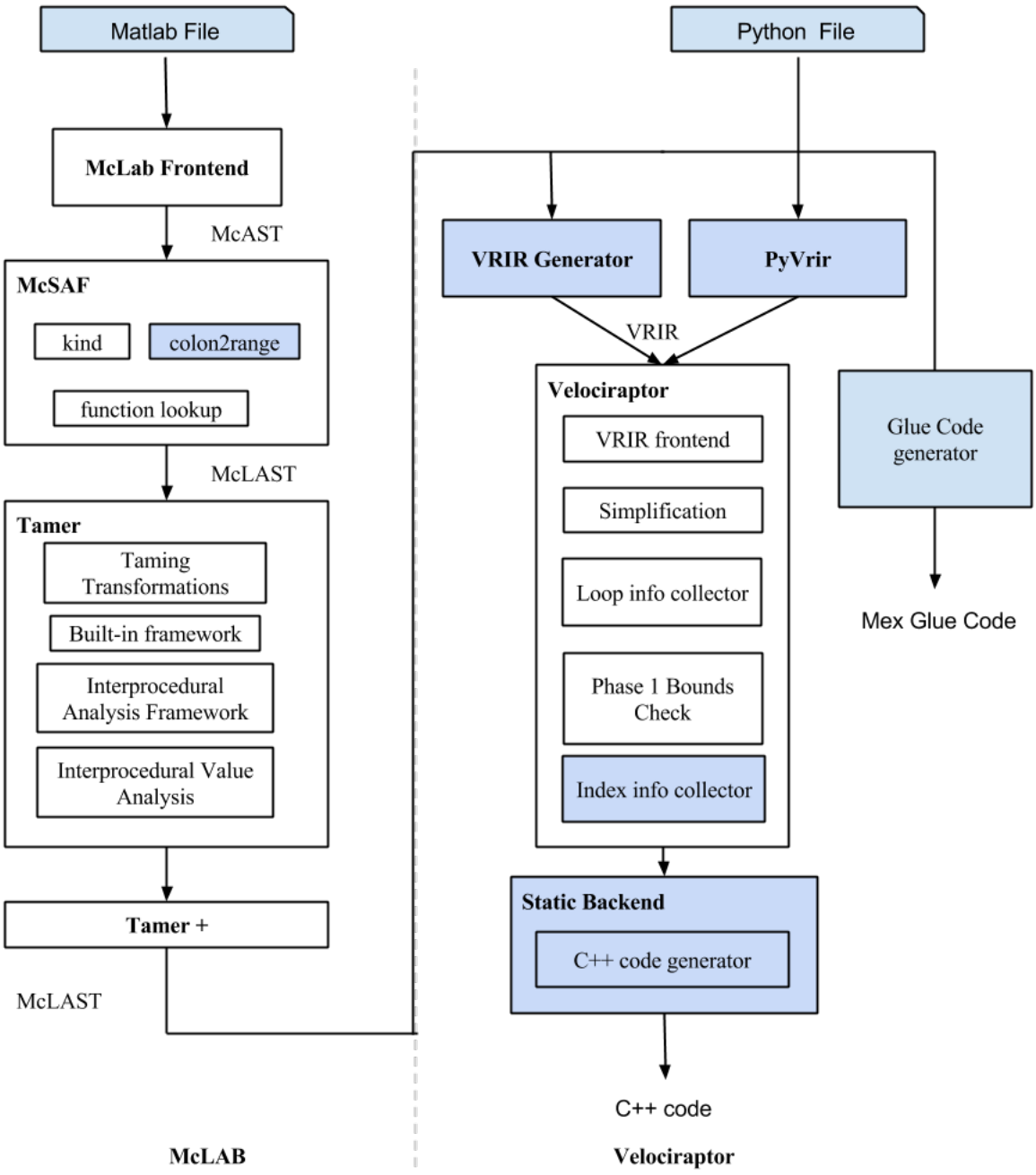

Figure 1.1 Overview of VeloCty. The shaded boxes indicate the components presented in this thesis. The other solid boxes correspond to existing *McLAB* and Velociraptor tools we use

of temporary variables and hence has better readability. The VRIR code generator takes McLAST as input and generates VRIR in the s-expression format. It also generates, glue code using the MATLAB MEX[Matc] API required for interfacing with MATLAB.

The VRIR is then parsed by the Velociraptor frontend and converted into an AST representation. Various passes such as the simplification pass, loop info collector and the index info collector pass are performed over the AST and is then passed to the static code generator. Finally, the code generator outputs C++ code which can then be compiled to a shared library along with the language specific run time library containing helper functions and the glue code.

1.2 The Execution Model

The Execution model, shown in *Figure 1.2*, describes how program execution occurs before and after statically compiling some of the methods in the program to C++ using Velo**C**ty. The user selects a function which he/she identifies as computationally intensive. Velo**C**ty generates a callgraph using the user-specified function as an entry point. In *Figure 1.2* the core1 funcrion is specified as the entry point. The compiler then generates a callgraph. In the current example, the callgraph contains core1 and core2. All functions in callgraph are then compiled to C++ by Velo**C**ty. The figure shows the implementation of the core1 function. The function contains a double array a and returns another double array textsfx. The array x is initialised inside the function using the builtin function zeros. The function also contains a for loop which iterates from 1 to 10. On each iteration, the i^{th} element of x is assigned the result of the sum of the i^{th} elment of a and a constant value 10 and a call is made to the core2 function.

The generated C++ code contains calls to functions in a language-specific library. The library contains functions which mirror the builtins in the source language. These functions are also written in C++ but are language-specific because the behaviour of the functions are dependent on the source language. In this case, the zeros function is implemented in the runtime library and the generated code contains a call to this function in the runtime library.

Velo**C**ty also generates glue code to interface with the source program. The MEX API and the Python/C[Foub] API are used for interfacing with MATLAB and Python respec-

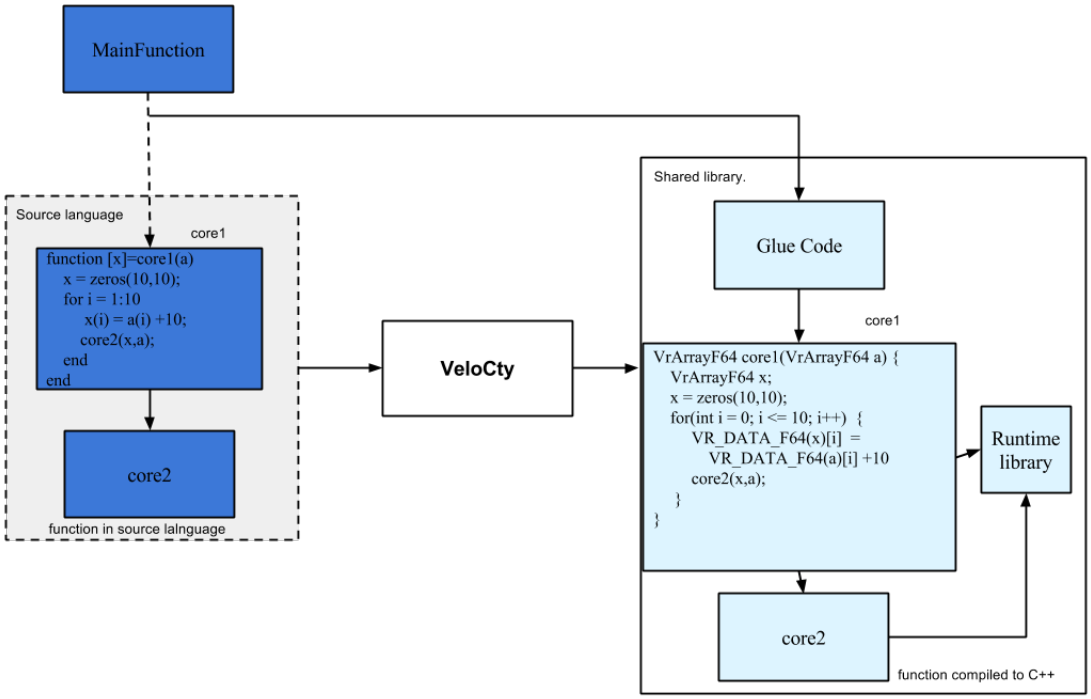

Figure 1.2 Execution model of VeloCty. The dark shaded blocks represent functions written in the source language and the blocks that are lightly shaded are the functions written in C++. The white block represents the Velo**C**ty compiler.

tively.

The generated code is compiled with the runtime and the glue code and packaged as a shared library. All calls to the entry point function, in this case, core1, would be directed to the shared library instead of the original source language version.

1.3 Contributions

The main contributions of this thesis are as follows.

- Design and implementation of a system for partially compiling array-based languages.

- Generating the Velociraptor intermediate representation from the McSAF intermediate representation.
- Implementation of a transformation from Colon expressions to Range expressions
- Generating glue code necessary for invoking C++ functions from MATLAB.
- Generating C++ code from the Velociraptor IR.
- Optimizing generated code by eliminating bounds checks removing unnecessary memory allocations and parallel code execution.

1.4 Thesis Outline

This thesis is divided into 9 chapters, including this one, which are structured as follows. *Chapter 2* gives a brief overview of the tools used by VeloCty. *Chapter 3* describes the translation from the McSAF intermediate representation to the Velociraptor intermediate representation(VRIR). *Chapter 4* talks about the generation of C++ code from VRIR. *Chapter 5* describes the various aspects of generating glue code for MATLAB's MEX API including how input data is converted from MEX data structures to VeloCty data structures. *Chapter 6* explains the code optimisations implemented to improve the performance. *Chapter 7* describes the performance of Velo**C**ty compared to various other systems. *Chapter 8* provides an overview of related work and *Chapter 9* concludes.

Chapter 2
Background

Velo**C**ty can compile functions written in MATLAB and NumPy to C++. In order to ensure that the code generated by Velo**C**ty matches the semantics of the language from which the code was generated, we first researched the semantics of the two languages. This chapter gives an overview of the semantics of the two languages that were important from the point of view of code generation. Additionally, we used C APIs provided by MATLAB, Python and NumPy to interface the generated code with the source language. The chapter also discusses the C APIs that were used by Velo**C**ty. Finally, the chapter describes the toolkits that were used by the compilation pipeline, namely the *Mc***LAB** toolkit and the Velociraptor toolkit.

2.1 Comparison of MATLAB and NumPy semantics

Our compiler supports two languages, namely MATLAB and NumPy. In order to ensure that the code generated matches the semantics of the language from which it was generated, we studied the semantics of both languages. This section discusses the similarities and differences of both languages.

2.1.1 MATLAB

MATLAB supports 1-indexing, that is the array index starts from 1. The array layout is always column major. The number of dimensions are greater than or equal to 2. Row and column vectors are 1xn and nx1 matrices respectively whereas scalars are 1x1 matrices. Function calls and array index operations have similar syntax as can be seen in Listing 2.1. Line 2 is an array index operation on A and line 3 is a call to the function sin.

```
1 A = zeros(3,3);
2 ... = A(2,3);
3 ... = sin(A);
```

Listing 2.1 An example of an array index operation and a function call.The array index operation and the function call have similar syntax.

MATLAB allows the number of indices to be greater than the number of dimensions of the array as long as the values of all indices in positions higher than the number of dimensions are one. Listing 2.2 gives an example of an array with two dimensions each of size three and an index operation on the array with four indices. As we can observe, all indices at positions greater than the number of dimensions, that is two, are one and hence the index operation is valid.

```
1 A = zeros(3,3);
2 ... = A(2,3,1,1);
```

Listing 2.2 An example of an array index operation where the number of indices are greater than the number of dimensions of the array

MATLAB also allows the number of indices to be less than the number of dimensions. We define this type of index operation as array flattening. Suppose we have an N-dimensional array A on which an index operation with K indices is performed. In this case, A can be treated as a K-dimensional array and the size of the K^{th} dimension can be defined as

$$Dim_K = \prod_{i=k}^{N} dim_i \tag{2.1}$$

where Dim_K is the new K$^t h$ and dim_i is the ith dimension of the original array A.
Note that the dimensions of the array are not permanently modified but only for the purpose of the index operation.

MATLAB also supports array slicing. Array slicing is an operation where a range of values can be provided in order to access a portion of the array instead of a single element. Listing 2.3 gives an example of the array slicing operation. The array A is 2-dimensional array with the sizes of the first and second dimensions as 4 and 3 respectively. The first index of the index operation on A contains a range with three values. The first value, 2 is known as the start value and gives the index from which the range starts. The second value is the step value, 2, which defines the interval between two indices and the third value is the stop value, 4, which defines the final index value. In this case the index values are 2 and 4 for the first index. In the case of the second index, the start, stop and step values are not provided. In this case, the start value is set to 1, the step value to 1 and the stop value to the size of the dimension. In this case, the second index refers to all the columns of the matrix. The index operation will return all the columns for the second and fourth row. The output will hence be a 2x3 matrix.

```
1 A = zeros(4,3);
2 ... = A(2:2:4,:);
```

Listing 2.3 An example of an array slicing operation in MATLAB

We define dimension-collapsing functions as those which perform an operation on a set of array elements and generate a single value. Thus the size of the output array differs from that of the input array. For example, the sum function in MATLAB when given as input a matrix, treats the matrix as a row of column vectors and returns a row vector, where each element is the sum of the column elements of each column of the matrix. Other examples of dimension collapsing functions are mean, max and min etc. These functions also accept an optional parameter as input which defines the dimension along which the array has to be collapsed.

```
1 import numpy
2 A = numpy.zeros([4,3]);
3 = ... A[2,1]
4 = ... A[2]
5 = ... A[-3,2]
6 = ... numpy.sin(A)
```

Listing 2.4 An example of indexing in NumPy

2.1.2 NumPy

NumPy supports 0-indexing, that is the array index starts from 0. It also supports negative indexing, that is the index values can be negative. In the case of negative indexing, the index starts from the end of the array. For example, an index value -1 refers to the last value of the dimension. The actual index value can be calculated by adding the size of the dimension to the index value. If the index value is -2 and the size of the dimension is n, the actual index is $n-2$. NumPy supports column major, row major and strided array layouts. Strided layouts allow users to define their own array layout scheme. A NumPy array has one or more dimensions. Additionally, NumPy differentiates between arrays and scalars.

Unlike MATLAB, Python does not allow the number of indices to be greater than the number of dimensions. The number of indices can be greater or less than the number of dimensions. If all the indices are numerical values and the number of indices are equal to the number of dimensions, a single value is returned. On the other hand, if the number of indices are less than the number of dimensions, a reference to the lower dimensions is returned. Line 3 in Listing 2.4 shows an index operation where the number of indices are equal to the number of dimensions. In this case, the element in the second row and first column is returned. On the other hand the index operation in line 4 contains one less index than the number of dimensions. In this case a reference to the second row is returned. Line 5 is an index operation with negative indexing. The first index -3 refers to the 1^{st} row of the array. Line 6 is a function call to the sin function. As we can see the syntax of the function call differs from that of an index operation which is unlike what we observe in MATLAB.

Slicing operations in Python are similar to those of MATLAB with the exception that the stop value is not included in the range of indices. Negative values in the ranges are also

```
1 import numpy
2 A = numpy.zeros([4,3]);
3 ... = A[3:1:-1,2]
```

Listing 2.5 An example of array slicing in NumPy

supported. Line 3 of Listing 2.5 gives an example of array slicing in NumPy. Rows 3 and then 2 are selected and from each row the element in the second column is returned. Note that the range is specified different from MATLAB. The step value is at the last position in the range whereas it is in between the start and stop values in MATLAB. In this example, the start value is 3 and the stop value is 1. Note that Python does not throw an error if the start or stop values exceed dimensions. The values are merely modified to the smallest or largest valid value depending on whether the lower and the upper bounds are exceeded respectively.

Dimension collapsing functions in NumPy behave differently from those in MATLAB. By default, the functions perform the operation on the entire array instead of only a set of elements. For example, the sum function will sum all the elements of the array and return a single value. If the output is to be collapsed against any dimension, the dimension has to be provided.

2.2 C APIs

We used the C APIs provided by MATLAB and Python to interface the generated code with the source language. To interface with MATLAB, we use the MEX[Matc] API and the Python/C[Foub] API is used for Python.

2.2.1 MEX

The Mathworks' MEX provides functions and data structures to allow interfacing code in MATLAB with code in C/C++. In order to use these functions, the mex.h header file is required to be included. A compiler to compile the C/C++ code is also provided. Data between the MATLAB code and the C++ code is passed as pointers to MxArrays. The

raw array data and meta data can be accessed through different MEX functions. MEX also provides functions to create and destroy MxArrays as well as other basic memory management functions. Table 2.1 gives a list of MEX functions that are used for interfacing with the source langauge.

MEX Functions	Description
mxCreateNumericArray	Creates an MxArray given dimensions, number of dimensions, element type and whether the array is real or complex
mxCreateDoubleScalar	Creates an MxArray with a single element of type double.
mxDestroyArray	Frees memory allocated to the MxArray
mxMalloc	Allocates memory of specified size in bytes
mxFree	Frees memory allocated by
mxGetData	Returns a void pointer to the raw array data of a MxArray
mxSetData	Sets the mxArray's data pointer to given memory.
mxSetDimensions	Sets the dimensions and number of dimensions of a mxArray
mxGetDimensions	Returns the dimensions of a mxArray
mxGetNumberOfDimensions	Returns the number of dimensions of a mxArray

Table 2.1 The table lists the MEX functions that were used by Velo**C**ty.

The entry point function for MEX has the name mexFunction. Every C/C++ program that is to be interfaced with MATLAB is required to have an implementation of the mexFunction. Listing 2.6 gives the function signature of the mexFunction. The function takes four input parameters. The first parameter, nlhs defines the number of output parameters. The second parameter plhs is an array of mxArray pointers. plhs holds the output parameters of the function. The third and fourth parameters, nrhs and prhs, define the number of input parameters and the input parameters of the function. All elements of the plhs array are set to NULL and hence mxArrays need to be created in the C/C++ code before the function returns. On the other hand, prhs contains the input mxArrays that have been created by the calling function.

The C/C++ program can then be compiled using the MEX compiler. The compiler

```
void mexFunction(int nlhs, mxArray *plhs[], int nrhs,
  const mxArray *prhs[])
```

Listing 2.6 Function signature of mexFunction

```
  static PyObject* arc_distance(PyObject* self, PyObject *args);
```

Listing 2.7 Signature of a function that can be called from Python

generates a dynamically linked library having the same name as the C/C++ source file. The MEX C/C++ function can then be called from a MATLAB program as a regular MATLAB function.

2.2.2 C APIs for Python

Programs written in C/C++ can be interfaced with Python using the Python/C[Foub] API. We also use the NumPy C-API[Com] passing and returning arrays. The data structures and functions provided to interface with Python can be used by including the header file Python.h. In order to interface with Python code, the C/C++ has to be structured as a Python module. A Python program passes data to the C/C++ program as a single parameter, which is a pointer to a PyObject. If multiple parameters are to be passed into the function, the PyObject may represent an array of other PyObjects. Unlike MEX, there is no restriction on the name of the entry point function. Listing 2.7 gives an example of a C function which serves as an entry point function. The static function, arc_distance, returns a pointer to a PyObject and take two input parameters, both PyObject pointers. The first input parameter, self is a pointer to the module object. The second parameter, args contains the input arguments of the function.

The methods that can be accessed from Python in the given module have to listed in an array of structs of the type PyMethodDef. PyMethodDef has four fields: method name as a string, pointer to the method implementation, METH_VARGS which tells Python how to access the method and finally the documentation for the method. The last entry into the array should have all NULL values to indicate the end of the array. Listing 2.8 gives an

example of the PyMethodDef array for the function arc_distance.

```
static PyMethodDef arc_distance_kernelMethods[] =
   {
        {"arc_distance",arc_distance,METH_VARARGS,"arc_distance of a circle. "},
        {NULL,NULL,0,NULL}
   };
```

Listing 2.8 An Example of the PyMethodDef struct

Additionally, a struct of type PyModuleDef also needs to be initialised. PyModuleDef describes the module. Listing 2.9 gives an example of the PyModuleStruct for the module arc_distance. The struct holds module information such as the name of the module, documentation and the PyMethodDef array among others.

```
static struct PyModuleDef arc_distance_kernelModule = {
      PyModuleDef_HEAD_INIT,
      "arc_distance_kernelModule",
      NULL,
      -1,
     arc_distance_kernelMethods,
 };
```

Listing 2.9 Example of the PyModuleDef struct

Finally, the C/C++ program is required to implement the module initialisation function. Listing 2.10 gives an example of the initialisation function. The name of the function is PyInit_<module name>, that is PyInit_ followed by the name of the module. In this case the name of the module is arc_distance. A module object is initialised using the method PyModule_Create which takes the PyModuleDef struct as input. The function import_array, is used to initialise NumPy specific constructs.

2.3 *McLAB*

McLAB is an extensible compiler toolkit for MATLAB. *McLAB* provides compilation, analysis and execution tools to optimise MATLAB. *McLAB* provides frameworks to aid

```
PyMODINIT_FUNC
  PyInit_arc_distance_kernel(void){
       PyObject* m = PyModule_Create(&arc_distance_kernelModule);
      import_array();
      return m;
  }
```

Listing 2.10 Example of the module initialisation function for the module arc_distance

static compilation of MATLAB programs to other languages such as FORTRAN and X10. These tools provide analyses which aid easy compilation of MATLAB programs to different targets. McSAF, Tamer and Tamer+ are the three frameworks that are used for implementing static compilers for MATLAB.

2.3.1 McSAF

McSAF is a static analysis framework for implementing static analyses for the MATLAB language. McSAF provides APIs and the core functionality to implement static analyses with ease. It also provides an intermediate representation known as McLAST on which the analyses and transformations can be performed. McLAST is a high-level AST[1] based representation with a structure close to the MATLAB program from which it was generated. McSAF can be used for various purposes such as static compilation to static and dynamic languages, code refactoring etc.

The kind analysis[DHR11], implemented using McSAF, separates array index operations from function calls. This analysis is crucial because both array index operations and function calls are syntactically similarly and hence can not be differentiated statically based on syntax alone.

The colon expression to range expression transformation was also performed using the McSAF framework. This transformation was a contribution on this thesis and is explained in Section 3.8.

Table 2.2 gives an example of a MATLAB function babai, and the equivalent McSAF code that was generated. As we can observe, the generated McSAF code is very close to

[1] Abstract Syntax Tree

the original MATLAB function. Through this framework we gather the information of the function calls and index operations of the function. Moreover, all the colon expressions have been converted to range expressions.

MATLAB	McSAF

`function z_hat = babai(R,y)`
`n=length(y);`
`z_hat=zeros(n,1);`
`z_hat(n)=round(y(n)./R(n,n));`
 	
`for k=n-1:-1:1`
`  par=R(k,k+1:n)*z_hat(k+1:n);`
`  ck=(y(k)-par)./R(k,k);`
`  z_hat(k)=round(ck);`
`end`
 	
`end`
	`function [z_hat] = babai(R, y)`
	`  n = length(y);`
	`  z_hat = zeros(n, 1);`
	`  z_hat(n) = round((y(n) ./ R(n, n)));`
	`  for k = ((n - 1) : (-1) : 1)`
	`  par = (R(k, ((k + 1) : n)) *`
	`  z_hat(((k + 1) : n)));`
	`  ck = ((y(k) - par) ./ R(k, k));`
	`  z_hat(k) = round(ck);`
	`  end`
	`end`

Table 2.2 The table gives an example of a MATLAB function babai and the generated McSAF code.

2.3.2 Tamer

Similar to McSAF, Tamer is an object oriented toolkit to implement analyses and transformations on MATLAB. Tamer facilitates the static compilation of MATLAB programs to different static languages.

Given an entry point function, Tamer generates a complete callgraph. It also handles the large number of MATLAB builtins through the Builtin framework.For every function in the callgraph, Tamer converts the function's McSAF intermediate representation, McLAST and generates Tame IR, a three address code[2] based intermediate representation with specialised AST nodes.

Tamer implements analyses on Tame IR which aids static compilation. These include the value analysis[DH12] which estimates MATLAB types, the shape analysis[LH14] which infers the dimensions of the variables and the IntegerOkay[Kum14] which identifies the variables having integer types.

Table 2.3 gives an example of a MATLAB function and the equivalent Tamer code. As we can see, since every statement is broken down into three address code, the length of

[2]In a three address code based IR, each statement as at most three operands

the generated Tame IR code is quite large. Moreover, we can also observe that operators like + and - have been replaced by function calls. This is because in MATLAB, operators are syntactic sugar and are replaced by function calls during execution. We get information about the type, the shape, that is the number of dimensions and the sizes of each dimensions of the variables in a function. We also get information about whether the variables are real or complex using the isComplex analysis[Kum].

MATLAB

```
function z_hat = babai(R,y)
n=length(y);
z_hat=zeros(n,1);
z_hat(n)=round(y(n)./R(n,n));

for k=n-1:-1:1
    par=R(k,k+1:n)*z_hat(k+1:n);
    ck=(y(k)-par)./R(k,k);
    z_hat(k)=round(ck);
end

end
```

McSAF

```
function  [z_hat] = babai(R, y)
  [n] = length(y);
  mc_t20 = 1;
  [z_hat] = zeros(n, mc_t20);
  [mc_t3] = y(n);
  [mc_t4] = R(n, n);
  [mc_t2] = rdivide(mc_t3, mc_t4);
  [mc_t0] = round(mc_t2);
  z_hat(n) = mc_t0;
  mc_t21 = 1;
  [mc_t18] = minus(n, mc_t21);
  mc_t22 = 1;
  [mc_t19] = uminus(mc_t22);
  mc_t25 = 1;
  for k = (mc_t18 : mc_t19 : mc_t25);
    mc_t10 = k;
    mc_t23 = 1;
    [mc_t12] = plus(k, mc_t23);
    mc_t13 = n;
    [mc_t11] = colon(mc_t12, mc_t13);
    [mc_t5] = R(mc_t10, mc_t11);
    mc_t24 = 1;
    [mc_t8] = plus(k, mc_t24);
    mc_t9 = n;
    [mc_t7] = colon(mc_t8, mc_t9);
    [mc_t6] = z_hat(mc_t7);
    [par] = mtimes(mc_t5, mc_t6);
    [mc_t16] = y(k);
    mc_t17 = par;
    [mc_t14] = minus(mc_t16, mc_t17);
    [mc_t15] = R(k, k);
    [ck] = rdivide(mc_t14, mc_t15);
    [mc_t1] = round(ck);
    z_hat(k) = mc_t1;
  end
end
```

Table 2.3 The Table gives an example of a MATLAB function babai and the generated Tame IR code.

2.3.3 Tamer+

Tamer+ is a code aggregation framework. Since Tame IR is a three address code based IR, the code generated from Tame IR is long and difficult for humans to read, due to the use of temporary variables. Hence, in order to improve code readability, Tamer+ aggregates multiple statements together and reduces the number of statements. Tamer+ takes as input Tame IR and outputs McSAF IR (McLAST). The generated code has fewer statements and is hence more readable. Tamer+ retains the information that was gathered through the analyses performed in Tamer. Tamer+ also generates a map from a sub-expression to its equivalent temporary if one exists. The type and shape information of the temporary would be the type and shape of the sub-expression. This map is therefore of importance to us as we will explain in the subsequent chapters.

Table 2.4 gives an example of a MATLAB function and the equivalent McSAF code generated from Tame IR by Tamer+. The code length is almost the same as the original MATLAB function.

MATLAB	McSAF(Generated by Tamer+)

`function z_hat = babai(R,y)`
`n=length(y);`
`z_hat=zeros(n,1);`
`z_hat(n)=round(y(n)./R(n,n));`
 	
`for k=n-1:-1:1`
`  par=R(k,k+1:n)*z_hat(k+1:n);`
`  ck=(y(k)-par)./R(k,k);`
`  z_hat(k)=round(ck);`
`end`
 	
`end`
	`function [z_hat] = babai(R, y)`
	`  [n] = length(y);`
	`  [z_hat] = zeros(n, 1);`
	`  z_hat(n) = round(rdivide(y(n), R(n, n)));`
	`  for k = (minus(n, 1) : uminus(1) : 1);`
	`  [par] = mtimes(R(k,`
	`  colon(plus(k, 1), n)),`
	`  z_hat(colon(plus(k, 1), n)));`
	`  [ck] = rdivide(minus(y(k), par)`
	`  , R(k, k));`
	`  z_hat(k) = round(ck);`
	`  end`
	`end`

Table 2.4 The Table gives an example of a MATLAB function babai and the generated McSAF code generated from Tame IR by Tamer+.

2.4 Velociraptor

Velociraptor is a compiler toolkit aimed at improving performance of array-based languages such as MATLAB and NumPy. The toolkit consists of an intermediate represen-

tation known as VRIR. The toolkit also provides various analysis on transformation on the IR. A compiler known as PyVrir, for Python to VRIR is also provided.

2.4.1 VRIR

VRIR is high-level strongly typed AST based intermediate representation. VRIR is designed to be flexible to accommodate semantics of different scientific languages such as MATLAB and Python's NumPy library. VRIR supports various array indexing schemes such as 0-indexing, 1-indexing and negative indexing and multiple array layouts such as row major, column major and stride major. VRIR also supports parallelism through constructs such as parallel for loop, map and reduce etc. VRIR can be generated as a string in the s-expression format which will then be converted to a C++ based AST. Velo**C**ty uses this approach in its compilation pipeline. Alternatively, the C++ AST can directly be generated.

2.4.2 Parser

If a language frontend compiling to VRIR, generates VRIR in the S-expression format and dumps it in a file, this file can be given to a parser implemented using ANTLR[Par]. The parser generates an ANTLR AST which is then converted into a C++ VRIR AST. This AST is then used for optimisations and code generation.

2.4.3 Analyses

The Velociraptor toolkit also performs analyses and optimisations on VRIR which can be resusable across compiler backends. The simplification pass simplifies expressions containing array operations into a three address code format. This simplification was useful to us while implementing the memory optimisation described in Section 6.3. The preliminary bounds check eliminations analysis, identifies and eliminates redundant bounds checks.

Chapter 3
Generating VRIR from the McSAF Intermediate Representation

As mentioned in the earlier chapters, Velo**C**ty supports MATLAB and Python's NumPy library. The Velo**C**ty backend takes VRIR as input and generates C++ code. We use PyVrir that is part of the Velociraptor toolkit generate VRIR from Python. However, no such tool exists to generate VRIR from MATLAB to VRIR. The *Mc***LAB** toolkit is a framework to aid static compilation of MATLAB to different languages. In order to support the compilation of MATLAB programs to C++ through Velo**C**ty, we implemented a VRIR generator using the *Mc***LAB** toolkit. Section 1.1 provided an overview of the compilation pipeline from MATLAB to VRIR and then to C++. As mentioned in the section, the VRIR generator takes an input McSAF IR and generates the S-expression version of VRIR.

VRIR generation had challenges. The McSAF IR is a MATLAB-specific IR whereas VRIR is designed to handle semantics of different languages and thus contains flags to specify semantic information such as array layout, indexing scheme etc. We had to ensure the appropriate flags were set to correctly represent the semantics of MATLAB. Moreover, VRIR is a strongly typed AST representation. Every expression node in VRIR has a type and shape information associated with it. McSAF does not explicitly hold this information and hence had to be determined during the compilation process. Additionally, MATLAB functions do not need an explicit return statement for the output. When a return statement is explicitly provided, the parameters that need to be returned are not specified. This is

because the output parameters are specified in the function signature. On the other hand, VRIR does not support output parameters and only supports output types. This difference in IR structure also had to be handled.

This chapter discusses the compilation of various nodes of the McSAF IR to VRIR, generation of the symbol table and how the types and shapes of expressions are determined.

3.1 Mapping types

In order to generate VRIR types, we require type and shape information as well as whether the symbol is real or complex. This information is obtained through the type analysis , the shape analysis and the isComplex analysis that were performed on Tamer. All variables and expressions are mapped to one of 5 VRIR types which are collectively known as VTypes.

3.1.1 Scalar Type

Scalar types are used for scalar symbols. In this case the shape of the symbol will have two dimensions each of size one. The scalar type also as a ctype flag which determines whether the symbol is real or complex. The symbol is considered to be real if the flag is set to zero and complex if it is set to one. The type of the data also needs to be specified in the generated VRIR. Types in MATLAB are known as MClasses. Table 3.1 gives a list of MClasses that are supported by the VRIR generator. Note that the VRIR generator does not support other MClasses such as char, unsigned integers and 16 and 8 bit integers.

MClass	VRIR Scalar Type	Description
Logical	bool	Boolean type
Int32	int32	32 bit Integer
Int64	int64	64 bit Integer
Float32	float32	32 bit Floating point
Float64	float64	64 bit Floating point

Table 3.1 The table lists the different MATLAB types known as MClasses that are supported by the VRIR generator and the Scalar types generated in VRIR.

Table 3.2 gives an example of VRIR for the scalar type from a MATLAB variable x.

The example describes a scalar symbol that is of type float64 and is real.

MATLAB	Generated VRIR
`x = 0;`	`(float64 :ctype 0)`

Table 3.2 The table shows an example of the generated scalar type for a scalar variable x in MATLAB.

3.1.2 Array Type

The array type is used to represent types for MATLAB arrays. Arrays can have two or more dimensions and at least one of the dimension sizes have to be greater than one. Note that although MATLAB considers scalars to be 1x1 matrices, we make the distinction between scalars and arrays. Shape information is used to determine whether the symbol is an array or a scalar. Array types of VRIR contain information about the number of dimensions of the array and the array layout. The array layout can be rowmajor, colmajor and strided. However, in case of MATLAB the layout is always colmajor. Array Types also contain a child node of scalar type. The scalar type holds information about the type of the array elements as well as whether they are real or complex. Table 3.3 gives an example of the generated array type. The example shows a variable that is assigned to a 3x3 matrix of using the zeros builtin function. The generated VRIR array type contains a ndims attribute which is set to 2 since there are two dimensions, and the array layout attribute is set to colmajor since all arrays in MATLAB are column major. Using the child scalar type node of the array type, we can determine that each element of the array is of type float64 and that each element is real.

MATLAB	Generated VRIR

`x = zeros(3,3);`	`(arraytype :layout colmajor :ndims 2`
	`  (float64 :ctype 0)`
	`)`

Table 3.3 The table shows an example of the generated array type for an array variable x in MATLAB.

3.1.3 Void Type

The void type is generally used as part of the Function type to convey the absence of the input or output parameters.

3.1.4 Function Type

Function types are associated with function definitions and function handles. They contain information about the types of the input and output parameters of the function. The functype node contains two child nodes, intypes and outtypes. Both nodes will have children that can be other VTypes such as scalar types, array types etc. The function types are part of the function node of VRIR. Table 3.4 gives an example of the Function type generated for the function babai. The function accepts two input parameters both of which are arrays and returns another array. The types of the input arguments are listed inside the intypes child whereas the output parameters are listed inside outtypes. Note that the body of the function is replaced by a statement inside chevrons which acts as a place holder.

MATLAB	Generated VRIR

`function [z_hat] = babai(R,y)`
`<Function Body>`
`end;`
	`(functype`
	`  (intypes`
	`  ( arraytype :layout colmajor :ndims 2`
	`  (float64 :ctype 0)`
	`  )`
	`  ( arraytype :layout colmajor :ndims 2`
	`  (float64 :ctype 0)`
	`  )`
	`  )`
	`  (outtypes`
	`  ( arraytype :layout colmajor :ndims 2`
	`  (float64 :ctype 0)`
	`  )`
	`  )`
	`)`

Table 3.4 The table shows an example of the generated func type for the function babai in MATLAB.

3.1.5 Tuple Type

Tuple types are used to define data structures which can have data of different types. Table 3.5 gives an example of the tuple type. The table shows a function call to spqr which has multiple returns. The function call expression as well as the expression on the LHS will both have tuple types. In this case the tuple type specifies that the two variables being returned have VTypes, scalar type and array type respectively.

MATLAB	generated VRIR

`[nc, r]  = spqr(a,tol,maxrc)`	`(tupletype`
	`  (float64 :ctype 0)`
	`  (arraytype :layout colmajor :ndims 2`
	`  (float64 :ctype 0)`
	`  )`
	`)`

Table 3.5 The table gives an example of a call to a function with multiple returns in MATLAB and the equivalent VRIR tuple type that is generated.

3.1.6 Domain Type

The domain type is associated with the domain expression explained in Subsection 3.6.6. Domain expressions themselves are only associated with For or parallel For loop statements. Domain types specify the types of all the iterator variables of the loop statements. The domain type has an attribute ndims which specifies the number of iteration variables of the loop. In case of MATLAB, there can only be one iteration variable per loop. Listing 3.1 gives an example of a domain type for a single iteration variable that is of type float64.

```
1 (domaintype :ndims 1 (float64 :ctype 0))
```

Listing 3.1 An example of the domain type in VRIR.

3.2 Symbol Table

The symbol table contains a list of symbols that are defined inside a function in VRIR. The table contains the name and the type of each symbol. Moreover, there is a unique

```
1  (symtable
2    (sym :id 5 :name par
3      (float64 :ctype 0))
4    (sym :id 0 :name R
5      ( arraytype :layout colmajor :ndims 2
6        (float64 :ctype 0)
7      )
8    )
9    (sym :id 4 :name k
10     (float64 :ctype 0)
11   )
12   (sym :id 3 :name n
13     (float64 :ctype 0)
14   )
15 )
```

Listing 3.2 Symbol table in VRIR

id associated with every symbol using which it is referenced in the function. There is a symbol table for every function in VRIR. Listing 3.2 gives an example of a symbol table. The symbol table contains a set of sym nodes each having a unique id. For example, the sym node with id 5 on line 2 is the symbol par which is of type float64. The VRIR generator adds symbols when it comes across new symbols while traversing the function's abstract syntax tree. The VRIR code for the symbol table is then generated after the function body.

3.3 Generating the Module VRIR node

The root node of VRIR is the module. Every valid VRIR must contain the module as its root node. The module contains an attribute, indexing, which defines the type of array indexing used. The attribute can have two values 0 indicating zero indexing and 1 indicating one indexing. Since the MATLAB arrays are one indexed, the indexing attribute is always set to 1. The module node also contains a name attribute specifying the name of the module. Additionally, it contains a fns child node which itself has multiple function nodes as its children. Listing 3.3 gives an example of the module node of VRIR. The name of the module is babai and the indexing attribute is set to one.

```
1  (module :name babai :indexing 1
```

```
  (fns
    <functions>
  )
)
```

Listing 3.3 The listing gives an example of a VRIR module that is generated by the VRIR generator.

3.4 Handling Functions

MATLAB programs can have one or more functions. As mentioned in Section 1.2, the user specifies the entry point function using which a callgraph containing functions that are reachable from the entry point function is generated. All of the functions that are part of the callgraph are compiled to VRIR. The function node in VRIR has multiple children all of which are required to generate the C++ code for the function.

- Name : The function name represents the name of the function.
- Arglist : The arglist is a list of integers which are the Ids of the input arguments in the symbol table.
- Func type : The Func type gives information about about the types and shapes of the input and output parameters of the function.
- Body : The body represents the body of the function. It consists of a list of statements.
- Symbol Table : Contains information about the symbols used in the function.

The Table 3.6 gives an example of the function VRIR node for the MATLAB function babai. The function has two input arguments and one output parameter. Thus the intypes has two Vtype nodes and the outtype has a single VType node. Moreover, Since there are two input arguments, the arglist has two arg nodes.

MATLAB	Generated VRIR

`function [z_hat] = babai(R,y)`
`<Function Body>`
`end;`
	`(function babai`
	`  (functype`
	`  (intypes`
	`  ( arraytype :layout colmajor :ndims 2`
	`  (float64 :ctype 0)`
	`  )`
	`  ( arraytype :layout colmajor :ndims 2`
	`  (float64 :ctype 0)`
	`  )`
	`  )`
	`  (outtypes`
	`  ( arraytype :layout colmajor :ndims 2`
	`  (float64 :ctype 0)`
	`  )`
	`  )`
	`  )`
	`  (arglist`
	`  (arg :id 0)`
	`  (arg :id 1)`
	`  )`
	`  (body`
	`  <body>`
	`  )`
	`  (symtable`
	`  <Symbol Table>`
	`  )`
	`)`

Table 3.6 The table shows an example of the generated Function VRIR node for the function babai in MATLAB.

3.5 Mapping statements

Many of the statements in MATLAB have equivalent VRIR statement nodes. However, some require additional processing while generating their VRIR equivalent.

3.5.1 Assignment Statements

Assignment statements in MATLAB are compiled to the assignment statement node in VRIR. The assignment statement node of VRIR contains two child nodes, lhs and rhs. As the names suggest, the left hand side expression of assignment statement in MATLAB is compiled to an expression inside the lhs node and the right hand side expression is compiled to an expression inside the rhs node. Table 3.7 gives an example of a MATLAB statement that is compiled to a assignment statement node in VRIR. The left hand side is a scalar

variable n and the right hand side is a call to the function length.

MATLAB	Generated VRIR
`n = length(y)`	`(assignstmt (lhs (name :id 3 (float64 :ctype 0))) (rhs (fncall :fnname length (float64 :ctype 0) (args (name :id 1 ( arraytype :layout colmajor :ndims 2 (float64 :ctype 0)))))))`

Table 3.7 The table shows an example of the generated assignment statement VRIR node a statement in MATLAB.

Copy Statements

We define copy statements as assignment statements where both the left hand side and right hand side are array variables. Listing 3.4 gives an example of a copy statement in MATLAB. An array B is copied into another array A. According to MATLAB semantics, a deep copy[1] has to be performed. However, VRIR supports a reference copy[2]. Hence an explicit copy function call has to be added. We make use the copy library function of VRIR that is explained in Subsection 3.6.2. The right hand side is added as an argument of the copy function and the function call itself becomes the rhs of the assignment statement[3]. Table 3.8 gives an example of the copy statement. The array A is copied to another array x. In the generated code a library call expression representing the call to the copy function on the rhs.

[1] The data is actually copied from one array to the other

[2] Only the reference of the array is copied to the other array. Thus both arrays are referring to the same data.

[3] As a future work we would like to implement an analysis to remove copies where they are not required.

```
1 B = zeros(3,3);
2 A = B;
```

Listing 3.4 The listing gives an example of a copy statement in MATLAB.

MATLAB	Generated VRIR
x=A;	(assignstmt (lhs (name :id 1 (arraytype :layout colmajor :ndims 2 (float64 :ctype 0)))) (rhs (libcall :libfunc copy (args (name :id 0 (arraytype :layout colmajor :ndims 2 (float64 :ctype 0)))))))

Table 3.8 The table shows an example of the generated copy assignment statement VRIR node a statement in MATLAB.

3.5.2 For and Parallel For Statements

The MATLAB For statement is mapped to the For statement node of VRIR and the Parfor statement to the parallel For in VRIR. The McSAF IR does not have a separate Parfor node. Instead the For statement node contains a boolean flag which when set to true implies that the node is a Parfor statement. The flag when set to false implies that the node is a for statement.

The For statement node in VRIR has 3 children. The body node represents the list of statements that make up the loop body. Itervars is an array of the symbol table ids of the iteration variables of the loops. In case of MATLAB, there are only be one iteration variable. The loopdomain contains a domain expression which in turn defines the bounds of the loop. Table 3.9 gives an example of the for statement in MATLAB and the equivalent

VRIR for statement node. The iteration variable is k and the loops bounds are n-1 to 1. The step value is -1.

MATLAB	Generated VRIR

`for k=n-1:-1:1`
`  <Loop Body>`
`end;`
	`(forstmt`
	`  (itervars`
	`  (sym :id 4`
	`  )`
	`  )`
	`  (loopdomain`
	`  (domain`
	`  ( domaintype :ndims 1 (float64 :ctype 0))`
	`  (range :exclude %0`
	`  (start`
	`  (minus`
	`  (float64 :ctype 0)`
	`  (lhs`
	`  (name :id 3`
	`  (float64 :ctype 0)`
	`  )`
	`  )`
	`  (rhs`
	`  (realconst :dval 1`
	`  (float64 :ctype 0)`
	`  )`
	`  )`
	`  )`
	`  )`
	`  (step`
	`  (negate`
	`  (float64 :ctype 0)`
	`  (realconst :dval 1`
	`  (float64 :ctype 0)`
	`  )`
	`  )`
	`  )`
	`  (stop`
	`  (realconst :dval 1`
	`  (float64 :ctype 0)`
	`  )`
	`  )`
	`  )`
	`  )`
	`  )`
	`  (body`
	`  <Loop Body>`
	`  )`
	`)`

Table 3.9 The table shows an example of the generated For statement VRIR node a statement in MATLAB.

The parallel for statement node in VRIR also contains the three child nodes mentioned above. However, it also contains an additional node, shared. The shared node contains a

list of symbol table ids of the variables that are shared across loop iterations.

3.5.3 Return Statement

The return statement in MATLAB is mapped to the return statement node in VRIR. However, the return statement in MATLAB and therefore the return statement node in McSAF IR, does not specify the variables to be returned. This is because the function node of McSAF IR contains the information about the output parameters. But the function node in VRIR does not have a child node for the output parameters. Hence to allow the VRIR backend to determine the variables that need to be returned, we explicitly add the output parameters specified by the McSAF IR function node to the return statement node of VRIR. Table 3.10 gives an example of the return statement. The MATLAB function has a single output parameter z_hat. However, the return does not specify the fact that z_hat is a return parameter. Hence it has to explicitly added, as we can observe in the VRIR code in the second column.

MATLAB	Generated VRIR

`function z_hat = babai(R,y)`
`  <Function Body>`
`  return;`
`end`
	`(returnstmt`
	`  (exprs`
	`  (name :id 2`
	`  (arraytype :layout colmajor :ndims 2`
	`  (float64 :ctype 0)`
	`  )`
	`  )`
	`  )`
	`)`

Table 3.10 The table shows an example of the generated Return statement VRIR node a statement in MATLAB.

In MATLAB, a function need not have an explicit return statement. All the output parameters are returned to the caller once the end of the function is reached. However, for reasons mentioned above, we need a return statement in VRIR. Hence a return statement is explicitly added along with the output parameters.

In some cases the return statement may not be accessible through all paths. For example, if a return statement is present inside an if block, the return statement will be executed only if the if condition is true. In such cases, we add the output parameters to the existing return statement and also add a return statement at the end of the function body.

Table 3.11 gives a list of possible cases for the return statement in a MATLAB function and the actions that are taken for each case.

Status of Return statement in MATLAB	Action taken
No Return statement present.	Statement explicitly added at the end of the function body along with return variables.
Return statement present. Not accessible from all paths	Statement explicitly added at the end of the function body along with return variables Return variables added to existing return statement.
Return statement present. Accessible from all paths	Return variables added to existing return statement.

Table 3.11 The table gives a list of possible cases for the presence of the return statement in a MATLAB function and the subsequent actions taken for each case.

3.5.4 If Statement

The If statement in MATLAB is compiled to the If statement node in VRIR. The If statement in VRIR has three child nodes. The test expression contains the If condition, the If child contains the list of statements inside the If block and the else child contains the list of statements inside the else block. Table 3.12 gives an example of the If statement.

MATLAB	Generated VRIR

`if <test condition>`
`  <If Block>`
`else`
`  <Else Block>`
`end;`
	`(ifstmt`
	`  (test`
	`  <test condition>`
	`  )`
	`  (if`
	`  <If block>`
	`  )`
	`  ( else`
	`  <Else block>`
	`  )`
	`)`

Table 3.12 The table shows an example of the generated If statement VRIR node a statement in MATLAB.

3.5.5 While Statement

Similar to the If statement, the While statement in MATLAB is mapped to the While statement node in VRIR. The While statement node in VRIR contains two child nodes. A test node while holds the While condition and the body node which holds the statements of the loop body.

MATLAB	Generated VRIR

`while <test condition>`
`  <While Body>`
`end;`
	`(whilestmt`
	`  (test`
	`  <test condition>`
	`  )`
	`  (body`
	`  <While Body>`
	`  )`
	`)`

Table 3.13 The table shows an example of the generated While statement VRIR node a statement in MATLAB.

3.5.6 Break and Continue statements

The break and continue statements in MATLAB are compiled to the break continue statement nodes in VRIR respectively.

3.6 Mapping Expressions

Similar to statements, many expressions in MATLAB have equivalent expression nodes in VRIR. However, every expression node must have a VType associated with it. This is not the case with the McSAF IR. Hence the VType of each expression is required to be calculated during code generation.

3.6.1 Name Expressions

Name expressions in MATLAB can either mean a variable or a call to a function with arguments. In the case of variables, a name expression is generated in VRIR. A function call expression is generated if the expression represents a call to a function. If expression is the

first occurrence of the variable, an entry in symbol table is also made. The name expression contains an id attribute. The id attribute value represents the id of the variable in the symbol table. Table 3.14 gives an example of a variable in MATLAB and its equivalent name expression node in VRIR. The example shows the generated name expression for variable A. The id of the variable in the symbol table is 10. The symbol table entry for the variable is also shown.

MATLAB	Generated VRIR
A	;; Generated Name expression (name :id 10) ;; Entry in symbol table. (sym :id 10 :name A (float64 :ctype 0))

Table 3.14 The table shows an example of the generated name expression node for a variable A in MATLAB. The entry of the variable in the symbol table is also shown.

3.6.2 Parameterized Expressions

Parameterized expressions can be mapped to many different nodes in VRIR depending on their semantics. In MATLAB, a parameterized expression can be an array index operation or a function call. The kind analysis[DHR11] is used to differentiate between function calls and array index operations.

Index Expressions

If the parameterized expression in McSAF represents an index operation, the VRIR generator compiles the expression to an index expression. Table 3.15 gives an example of an array index operation in MATLAB and the equivalent VRIR code that was generated. The index operation has two indices. The first one is a simple numeric index k, whereas the second one is a slice index and hence specifies a range of index values starting from k+1 to n. The VRIR index expression node has an arrayid attribute which holds the id of the array inside the symbol table. In the example, the array id 0 refers to the array R. The copyslice flag whether the values of set of indices represented by the indices have to be copied to a new

array. The indices child node of array holds the set of indices of the index operation. Each child node is of type index. Note that this node is different from the index expression node. The index node has two attributes. The boundschecks attribute indicates whether array bounds checks need to be added for the index. The negative attribute indicates whether the index value can be negative. In case of MATLAB, the boundscheck attribute is set to %1 to include bounds checks and the negative attribute is set to %0 to indicate that negative indexing is not supported.

MATLAB	Generated VRIR

`R(k,(k+1):n)`	`(index :arrayid 0 :copyslice %1`
	`( arraytype :layout colmajor :ndims 2`
	`(float64 :ctype 0))`
	`(indices`
	`(index :boundscheck %1 :negative %0`
	`(name :id 4`
	`(float64 :ctype 0)`
	`)`
	`)`
	`(index :boundscheck %1 :negative %0`
	`(range :exclude %0`
	`(start`
	`(plus`
	`(float64 :ctype 0)`
	`(lhs`
	`(name :id 4`
	`(float64 :ctype 0)`
	`)`
	`)`
	`(rhs`
	`(realconst :dval 1`
	`(float64 :ctype 0)`
	`)`
	`)`
	`)`
	`)`
	`(stop`
	`(name :id 3`
	`(float64 :ctype 0)`
	`)`
	`)`
	`)`
	`)`
	`)`
	`)`

Table 3.15 The table shows a MATLAB index operation with a slice operation that is compiled VRIR

Function Call Expressions

Parameterized expressions that are calls to functions in MATLAB can be divided into four broad categories: operators, library calls, allocation function calls and miscellaneous function calls.

Operators include binary and unary operators such as plus, minus, unary minus etc. Although the McSAF IR does have nodes for all the operators, Tamer converts the operators to function calls when generating TameIR and Tamer+ keeps them as function calls. In case of operations on scalars, we convert the parameterized expressions representing operators to equivalent operators in VRIR. For some of these operators, if at least one of the operands are arrays, a libcall expression is generated. Table 3.16 gives a list of MATLAB operators and the VRIR nodes that are generated. If the last column, array operands, has a 'yes', the VRIR node is also generated for the array operands.

Matlab function	VRIR Node	Array Operands
plus	plus	No
minus	minus	No
rdivide	div	No
mtimes	mmult	No
times	mult	No
or	or	Yes
eq	eq	Yes
le	leq	Yes
ge	geq	Yes
lt	lt	Yes
gt	gt	Yes
uminus	negate	Yes
not	negate	Yes

Table 3.16 The table list the operators in MATLAB and the equivalent VRIR nodes that are generated.

Table 3.17 gives an example of a plus operator in MATLAB which has two operand expressions that are converted to the plus expression in VRIR. All other operators have a similar structure in VRIR.

As mentioned, operations on arrays are compiled to library call operations in VRIR.

MATLAB	Generated VRIR
<op1> + <op2>;	(plus (float64 :ctype 0) (lhs <op2>) (rhs <op2>))

Table 3.17 The table shows an example of a plus operator in MATLAB that is converted to a plus expression node in VRIR

Additionally, operators that can only take array operands such as matrix multiplication, transpose, matrix division among others, are also supported through library call expressions. Library call expressions also support some other functions that are commonly used in numerical and scientific computing and hence many of those functions are also compiled to library call expressions. Table 3.18 lists the scientific functions that are supported by the libcall expression.

Calls to functions like zeros and ones which are used to create arrays in MATLAB are compiled to the alloc expressions in VRIR. Table 3.19 gives an example of a MATLAB function zeros and the VRIR alloc expression that was generated. The alloc expression contains a func attribute which defines the name of the function. It takes three values, zeros, ones and empty. The zeros function creates an array and initialises all elements to zero, ones creates an array and initialises all elements to one and empty creates an uninitialised array. MATLAB does not support the empty function and hence the VRIR generator only generates the ones and the zeros function. The alloc expression also has a child node args which holds the input arguments of the function.

Function calls which do not qualify as library call expressions or alloc expressions are compiled to the function call expression node in VRIR. These include calls to user-defined functions as well as builtin functions that are not supported by alloc or library call expressions.

In MATLAB arguments to functions are passed by value. On the other hand, in VRIR, arguments are passed by reference. Hence in order to generate code that matches MATLAB semantics, we add calls to the library call function copy for every array argument that is

Library Functions	Description
Sqrt	Square root
Log2	Log with base 2
Log10	Log with base 10
Expe	exponent of e
Exp10	exponent of 10
Sin	Trigonometric Sin
Cos	Trigonometric Cosine
Tan	Trigonometric tangent
Asin	Inverse sin
Acos	Inverse cosine
Atan	Inverse tangent
Pow	power function
Sum	Sum function
Prod	Product function
Atan2	Arc Tangent
Abs	Absolute value
Min	Min function
Max	Max Function
Mean	Mean function
Copy	Copy function
Mmult	Matrix Multiplication
Mrdiv	Matrix right division
Mldiv	Matrix left division
Div	Element wise array division
Mult	Element wise array multiplication
Plus	Element wise array addition
Minus	Element wise array subtraction

Table 3.18 The table lists the functions supported by the library call expression in VRIR.

MATLAB	Generated VRIR

`zeros(m,n);`	`(alloc :func zeros`
	`  ( arraytype :layout colmajor :ndims 2`
	`  (float64 :ctype 0)`
	`  )`
	`  (args`
	`  (name :id 2`
	`  (float64 :ctype 0)`
	`  )`
	`  (name :id 4`
	`  (float64 :ctype 0)`
	`  )`
	`  )`
	`)`

Table 3.19 The table gives an example of the zeros function call in MATLAB and the equivalent alloc expression that is generated in VRIR.

passed to a call to a user-defined function. Builtin implementations ensure that a input arguments are copied if they have to be written to and hence no function calls to copy are generated. Table 3.20 gives an example of a user-defined function gauss that is compiled to a function call expression in VRIR. The name of the function is defined by the attribute fnname. The args child node contains the input arguments to the function. The arguments are copied by adding a call to the library call function copy.

MATLAB	Generated VRIR

`gauss(n,m)`	`(fncall :fnname gauss`
	`  (float64 :ctype 0)`
	`  (args`
	`  (libcall :libfunc copy`
	`  (float64 :ctype 0)`
	`  (args`
	`  (name :id 4`
	`  (float64 :ctype 0)`
	`  )`
	`  )`
	`  )`
	`  ( libcall :libfunc copy`
	`  (float64 :ctype 0)`
	`  (args`
	`  (name :id 11`
	`  (float64 :ctype 0)`
	`  )`
	`  )`
	`  )`
	`  )`
	`)`

Table 3.20 The table gives an example of a user-defined function call in MATLAB and the equivalent function call expression that is generated in VRIR.

3.6.3 Matrix Expressions

Matrix expressions in MATLAB are used to represent multiple expressions and are often found on the left hand side of an assignment statement where the right hand side is a call to a function with multiple output parameters. Matrix expressions are compiled to tuple expressions in VRIR. Table 3.21 gives an example of a matrix expression and the equivalent tuple expression. A tuple type is generated for a tuple expression which holds the types for each of the expressions inside the tuple expression. The tuple expression also holds a elem child node which holds the expressions of the matrix expression. In this case, the matrix expression contains two name expressions. Note the generated VRIR code only depicts the left hand side of the assignment statement.

MATLAB	generated VRIR

`[nc, r]  = spqr(a,tol,maxrc)`	`(tuple`
	`  (tupletype`
	`  (float64 :ctype 0)`
	`  (arraytype :layout colmajor :ndims 2`
	`  (float64 :ctype 0)`
	`  )`
	`  )`
	`  (elems`
	`  (name :id 3`
	`  (float64 :ctype 0)`
	`  )`
	`  (name :id 8`
	`  ( arraytype :layout colmajor :ndims 2`
	`  (float64 :ctype 0)`
	`  )`
	`  )`
	`  )`
	`)`

Table 3.21 The table gives an example of a matrix expression in MATLAB and the equivalent VRIR tuple expression that is generated.

3.6.4 Literal Expressions

Literal expressions are expressions in MATLAB holding constant value. There are three types of literal expression in MATLAB: FP literal expressions which represent the floating point constants, Int literal expressions which represent integer constants and string literal expressions which represent strings. Since VRIR does not support strings, the VRIR gen-

erator does not support the string literal expression. Both the Fp literal expression and the Int literal expressions are compiled to the constant expression in VRIR.

Table 3.22 gives an example of the constant expression in VRIR that is generated from a constant value in MATLAB. The constant expression in VRIR has a dval attribute which specifies a floating point constant value. Whether the value is 64 bit or 32 bit can be determined by checking the type of the expression. In case of Int literal expressions, the ival attribute is used.

MATLAB	generated VRIR

`oldcap = 0;`	`(realconst :dval 0`
	`  (float64 :ctype 0)`
	`)`

Table 3.22 the table gives an example of a FP literal expression in MATLAB and the equivalent VRIR constant expression that is generated.

3.6.5 Range Expressions

Range expressions are used to define a range of values. They hold three expressions, start, stop and step. The start expression refers to the start of the range, the stop to the end of the range and the step refers to the interval between two consecutive values. A range expression is compiled to a range node in VRIR. A range node also has three expressions, start, stop and step. The range node represents a range from the start expressions value to the stop expression value with intervals of the step expression value. Whether the stop expression value is included in the range is determined using the exclude attribute. If the exclude attribute is set to %1 the stop expression value is excluded and the stop expression value is included when the exclude attribute is set to %0. In case of MATLAB, since the stop expression value is always included, the exclude attribute is always set to %0. The step expression value is optional and defaults to 1 if not specified. Ranges are used for two reasons, one to represent loop bounds and other to represent an array slice in an index operation. Listing 3.5 gives an example of a range node in VRIR. The exclude flag is set to %0 and hence the stop expression value will be included.

```
(range :exclude %0
  (start
    <Start Expression>
  )
  (step
    <Step Expression>
    )
  (stop
    <Stop Expression>
  )
)
```

Listing 3.5 Example of a Range in VRIR

3.6.6 Domain Expressions

Domain expressions are used in for statements to specify the loop bounds. Domain expressions can support multiple loop bounds, one for each iteration variable. However MATLAB only allows a single iteration variable for a loop and hence only one set of loop bounds exist inside a domain expression for MATLAB. The VType of the domain expression is the domain type which holds the VTypes of all the iteration variables of the loop. The loop bounds are represented by ranges described in Subsection 3.6.5.

Table 3.23 gives an example of the domain expression that is generated as part of the for statement in VRIR. The Domain expression has a domain type and a single range for the iteration variable. The range starts from 1 and stops at na. Since the exclude attribute is not set, the stop value is included in the range.

3.7 Determining VTypes of expressions

The Tamer framework provides analyses such as the value analysis, shape analysis and the isComplex analysis. For every name expression in a function, these analyses determine the variable's type, shape and whether it is real or complex. This information is required in order to generate the VType of an expression. We also need to use the map between expressions in McSAF and their equivalent temporaries in TameIR that is provided by Tamer+ to determine VTypes of expressions other than the name expression.

MATLAB	Generated VRIR

`for ii = 1:na`
`<Loop Body>`
`end;`
	`(domain`
	`( domaintype :ndims 1`
	`(float64 :ctype 0)`
	`)`
	`(range :exclude %0`
	`(start`
	`(realconst :dval 1`
	`(float64 :ctype 0)`
	`)`
	`)`
	`(stop`
	`(name :id 0`
	`(float64 :ctype 0)`
	`)`
	`)`
	`)`
	`)`

Table 3.23 The table gives an example of a for statement in MATLAB and the domain expression that is generated as a part of the for statement in VRIR.

3.7.1 Determining type of name expressions

Name expressions store the name of the variable as a string. The variable name can be used to access the information required for generating VTypes stored in the value, shape and isComplex analyses. Thus in case of name expressions, the VType can be determined directly using the analyses.

3.7.2 Determining VTypes of Other Expressions

An expression in McSAF IR, generated by Tamer+ can be classified into two broad categories. One which was part of a statement in three address code in the original MATLAB function and one which was part of statement that was broken down into multiple three address code statements by Tamer. In the first case, Tamer+ does not aggregate multiple statements whereas it does aggregate multiple statements in the second case.

In the first case, if the expression is on the LHS of the statement, the expression will be a name expression and hence its VType can be determined using the method for name expressions mentioned above. If the expression on the RHS, we calculate the VType of the LHS expression which will also be the VType of the RHS expression. *Figure 3.1* gives an example of this case. Two variables A and B are multiplied and the result is assigned to C.

In this case, Tamer does not break down the statement into multiple statements and hence no temporaries are generated. The type and shape information of the name expressions, A, B and C, can be determined by using the analyses directly. However, in case of the multiplication expression, its type, shape and whether it is complex or real can be determined by looking at the expression on the LHS, C, and assigning C's information to the multiplication expression.

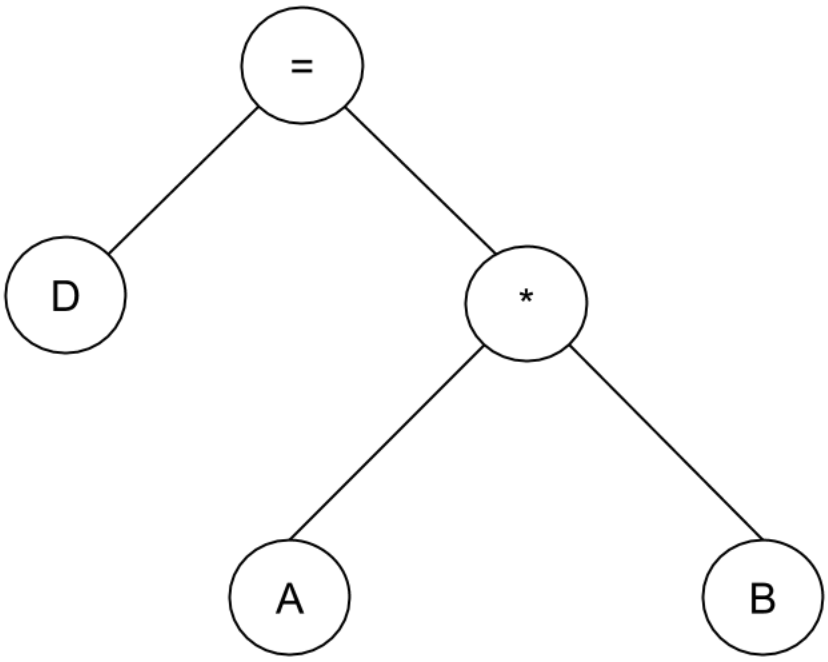

Figure 3.1 The figure gives an example of an statement in MATLAB which is already in three address code and hence is not broken down by Tamer

In the second case, if the expression is on the LHS, the expression is either a matrix expression or a parameterized expression. In such cases the RHS expression has a temporary

variable that is associated with it. The VType for the temporary variable can be generated which will be the VType for the LHS expression. If the expression is on the RHS, the expression itself will have a temporary variable associated with it. *Figure 3.2* gives an example of a statement which would be broken down into multiple statements by Tamer. As

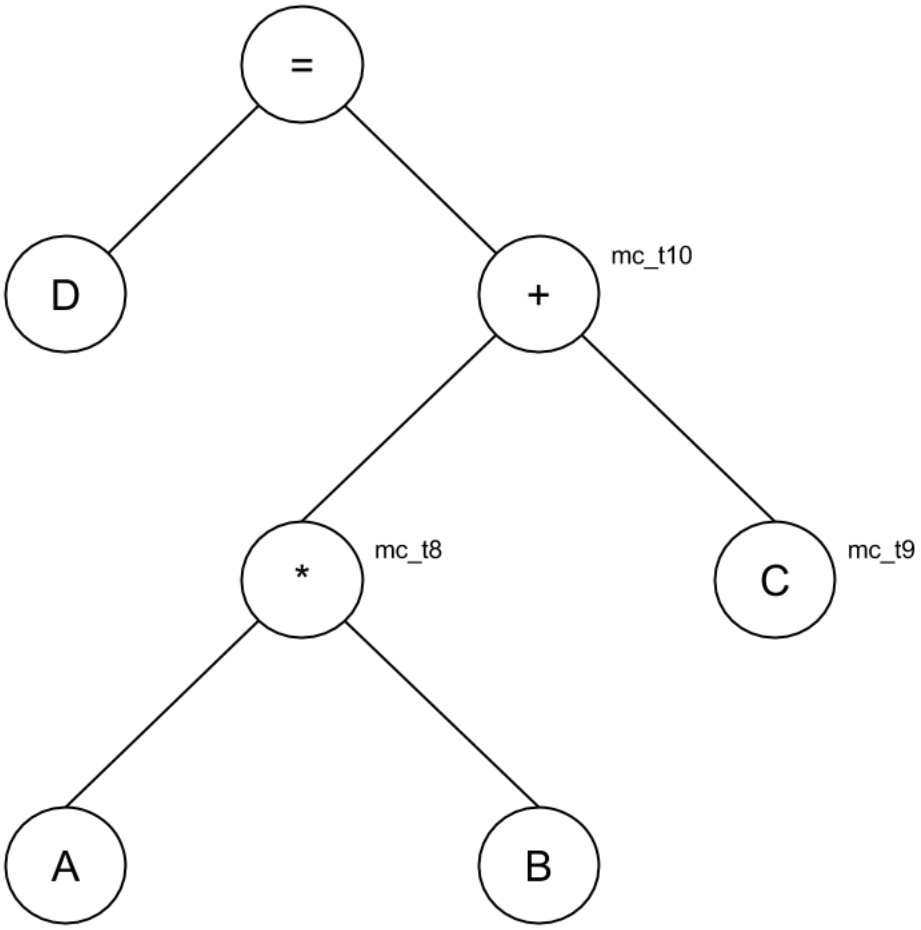

Figure 3.2 The figure gives an example of an statement in MATLAB which is not in three address code and hence is broken down by Tamer using temporaries

we can observe each sub-expression has a temporary variable to which it is assigned. The output of the plus expression is assigned to the variable mc_t10 and the output of the multiplication expression is assigned to mc_t8. The type, shape and whether the expressions are real or complex can be determined by fetching the same information for the temporaries

from the analyses.

3.8 Colon Expression transformation

The colon expression in MATLAB is used in index operations when all the elements of one or more dimensions have to be specified. Listing 3.6 gives an example of an index operation with a colon expression. The second index of the index operation on array A, is a colon expression. The colon expression selects all the columns of the array A. For every column index, the element in row 1 is selected. Thus, the index operation will fetch that all values from all the column that are in the first row. Note that we are assuming that the A is a matrix. If the number of dimensions are greater than 2 (greater than the number of indices), the stop value for the colon expression is the product of all the dimension sizes starting from the second dimension. We call this as array dimension flattening.

```
1 ... = A(1,:);
```

Listing 3.6 An example of the an array index operation with a colon expression as an index.

In this case, if there are 3 dimensions in total, the stop value of the colon expression will be the product of the sizes of the second and the third dimensions.

There is no equivalent statement in VRIR. In order to generate a range, we need the start and stop values. The start value will always be one. In order to fetch the stop value, we implemented a transformation which inserts the code to calculate the stop value of the colon expression.This transformation takes into account that an dimensions will have to be flattened if the the colon expression appears as the last index of of the array index operation.

We insert a statement before the index operation which stores the size of the dimensions for which the colon expression appears. If the colon expression appears as the last index of the array index operation, we also insert a for loop which takes a product of all the dimension sizes starting from the index position where the colon expression appears to the last dimension of the array. The colon expression is then replaced by range expression with the start value as 1 and the stop value as the calculated size. The range expression can then be compiled to the range described in Subsection 3.6.5. Table 3.24 gives an example of

how the McSAF IR is transformed. The first column shows the pretty printed McSAF code before the transformation and the second column shows the pretty printed McSAF after the transformation. The code contains three array index operations, each of which have a colon expression. Since for all three index operations the colon expression appears as the last index, a for loop is also inserted which flattens the array dimensions should the number of dimensions be greater than the number of indices. Moreover, the colon expression is replaced by the range expression with the start value as one and the stop value the one which was calculated.

McSAF (Before transformation)	McSAF (After transformation)

`dr(:) = (R(jj,: ) - R(ii, :));`	`dim_temp4 = 1;`
	`for dim_temp5 = (1 : ndims(dr));`
	`  dim_temp4 = (dim_temp4 *`
	`  size(dr, dim_temp5));`
	`end;`
	`dim_temp6 = 1;`
	`for dim_temp7 = (2 : ndims(R));`
	`  dim_temp6 = (dim_temp6 *`
	`  size(R, dim_temp7));`
	`end;`
	`dim_temp8 = 1;`
	`for dim_temp9 = (2 : ndims(R));`
	`  dim_temp8 = (dim_temp8 *`
	`  size(R, dim_temp9));`
	`end;`
	`dr((1 : dim_temp4)) = (R(jj, (1 : dim_temp6)) -`
	`  R(ii, (1 : dim_temp8)));`

Table 3.24 The table gives an example of the transformation from the colon expression to a range expression.

Chapter 4
Generating C++ from VRIR

An important contribution of the thesis is the static generation of C++ code from VRIR. Due to differences in the semantics of VRIR and C++, we faced various challenges during code generation. As described in *Chapter 2*, VRIR is a high level strongly typed AST designed to support easy compilation of a wide range of array-based languages. Hence, it supports different indexing schemes such as 0-indexing, 1-indexing and negative indexing as well as different array layout schemes such as row-major and column-major. C++ on the other hand does not have an built in support for array operations, and only supports 0-indexing and a row major layout. Moreover, VRIR also supports multiple returns. On the other hand, we can only return a single value, which can be a scalar, class, struct or a pointer, in C++. This chapter describes the runtime library and how different nodes of VRIR such as statements, types and expressions are compiled to C++ constructs.

4.1 Runtime library

Languages like MATLAB and Python's NumPy library provide a number of high-level numerical and scientific functions. These functions include trigonometric functions such as sin, cos etc. memory allocation functions such as zeros, ones among others. Moreover, MATLAB and NumPy also provide simple arithmetic operations on arrays such as multiplication, addition, transpose etc. Additionally, these languages also implicitly provide bounds checks for indexing operations. C++ on the other hand, does not provide many

```
typedef struct VrArrayF64{
  double *data;
  dim_type* dims;
  int ndims;
}VrArrayF64;

typedef struct VrArrayF32{
  float *data;
  dim_type* dims;
  int ndims;
}VrArrayF32;

typedef struct VrArrayI32{
  int *data;
  dim_type* dims;
  int ndims;
}VrArrayI32;

typedef struct VrArrayI64{
  int *data;
  dim_type* dims;
  int ndims;
}VrArrayI64;
```

Listing 4.1 Structure of VrArrays for real data

of the functions mentioned above. Hence we provide a language specific runtime library to implement these functions. We currently provide libraries for MATLAB and Python. Simple arithmetic operations on arrays are provided using BLAS libraries. We use the Intel MKL library for MATLAB and the OpenBLAS library for Python. The libraries also provide implementations for VrArrays, a VeloCty specific array representation.

4.1.1 VrArrays

Unlike array-based languages, C++ arrays do not store additional information such as the number of dimensions or the sizes of each dimensions. This information is useful while performing various operations such as multiplication, addition etc and hence it was necessary for us to store it. One solution was to store this information separately. However, this approach increases the number of parameters that need to be passed to functions implementing array operations. Moreover, when assigning to an array additional code that needs to be generated to update the dimension sizes and the number of dimensions. Hence, we implemented structs for arrays of all the data types supported by VRIR. We call the structs

```
typedef struct VrArrayCF32{
  float complex *data;
  dim_type* dims;
  int ndims;
}VrArrayCF32;

typedef struct VrArrayCF64{
  double complex *data;
  dim_type* dims;
  int ndims;
}VrArrayCF64;
```

Listing 4.2 Structure of VrArrays for complex data

collectively as VrArrays.

VrArrays are represented as C++ structs and encapsulate array data as well as the meta-data. They contain a pointer to the data as well as other necessary information such as the number of dimensions and the size of each dimension. Listing 4.1 gives the structures of the VrArrays representing real data and Listing 4.2 lists the VrArrays representing the complex data. Each VrArray has a data field which is a pointer to the array data. The type of the data field depends on the type of the VrArray. For example, the type of VrArrayF64, which is used to represent an float64 array is double.

There are separate VrArray types for complex and real arrays of the same type. All operations on arrays in the language runtime take VrArrays as input. This allows single parameter to be passed for array instead of passing the data, dimensions and number of dimensions separately. The full list of VrArrays and the types of their corresponding data fields is given in Table 4.1.

4.1.2 Memory allocation functions

Memory allocation functions are used to create n-dimensional arrays. The size of each dimension has to be provided as input. The compiler generates these functions from the alloc expression in VRIR. There three types of memory allocation functions.

- zeros : Allocates memory for a n-dimensional array and initialises all elements in the array to zero.

VrArray Type	Data Field Type
VrArrayF64	double
VrArrayF32	float
VrArrayCF64	double complex
VrArrayCF32	float complex
VrArrayI64	long
VrArrayI32	int
VrArrayB	bool

Table 4.1 Data field types of different VrArrays. Table depicting the types of the data field for different VrArray types.

- ones : Allocates memory for a n-dimensional array and initialises all elements in the array to one.
- empty : Allocates memory for a n-dimensional array but does not initialise the array.

Every function type has different implementations for different array types. The function name is as follows, <functionType>_<array type>. Table 4.2 gives an example of a memory allocation function. In the example a zeros function for create a double array, VrArrayF64, is created. The array has two dimensions of size m and n respectively. Note that the empty function is supported in Python but not supported in MATLAB. Hence only the Python runtime library supports it.

VRIR	Generated C++

`(alloc :func zeros`
`  ( arraytype :layout colmajor :ndims 2`
`  (float64 :ctype 0)`
`  ) (args`
`  (name :id 2`
`  (float64 :ctype 0)`
`  )`
`  (name :id 4`
`  (float64 :ctype 0)`
`  )`
`  )`
`)`
	`zeros_double(2,(int)m,(int)n)`

Table 4.2 The table shows an example of an alloc expression in VRIR that is converted to a zeros function call in C++

4.1.3 Mathematical functions

The runtime library supports various mathematical functions that can be found in the high-level languages. These include trigonometric operations, exponential functions etc. Many of these functions can work on both scalars and arrays. For scalars, we generate calls to functions in the standard C++ library. For arrays, calls to functions in the runtime library are generated. These functions can be generated from both libcall expressions as well as function call expressions. Functions on arrays can be divided into two types, element-wise functions and dimension collapsing functions.

Element-wise functions

These functions operate on each element of the array independently. The dimensions of the output array are the same as that of the input array. A few examples of these functions include sin, cos, exp, etc.

Dimension collapsing functions

These functions combine multiple array elements to generate a output. The dimensions of the output array are not the same as the input array. Sum,mean, prod and dot are a few examples of dimension collapsing functions. The dimensions of the output of these functions are many times dependent on the dimensions of the input array. For example, in MATLAB, if a matrix is given as an input to the sum function, the function will calculate the the sum of each column and return a row vector. On the other hand, if a row or column vector is provided as input, the sum function will calculate the sum of all the elements and return a scalar value. Additionally, in Python, the sum function will calculate sum of all the elements in the array and return a scalar value by default. The function will return a vector if an additional argument specifying the dimension along which the sum has to be calculated is provided. Taking into account these differences, we provide two sets of functions, one for cases where a scalar value is returned and one where an array is returned. The names of the functions which return scalars have the word 'scalar' appended to the names of the original function. For example, if the name of the original function which returns an array

is sum, the name of the function returning a scalar value is sum_scalar.

4.1.4 Array Operations

As mentioned before, C++ does not support basic operations such as addition, multiplication or transpose on arrays. Hence we support these operations through the runtime library. The operations are implemented as static methods of a class. There is a class for every array type. For example, the operations for VrArrayF64 are implemented as static methods for the class BlasDouble. Table 4.3 gives the entire list of classes implementing array operations for VrArrays. The methods call BLAS functions where possible for improved performance. We have also implemented specialised versions of many of the methods for memory allocation optimisation that is described in Section 6.3. Table 4.4 gives a list of

VrArray	Class name
VrArrayF64	BlasDouble
VrArrayF32	BlasSingle
VrArrayCF64	BlasComplexDouble
VrArrayCF32	BlasComplexSingle
VrArrayI64	BlasLong
VrArrayI32	BlasInt

Table 4.3 The table gives a list of VrArrays and the respective names of classes implementing array operations

array operations implemented by the runtime library.

4.2 Mapping Types

Data types in VRIR, known as VTypes, can be categorized into 5 types, namely Scalar type, Array type, Void type, Domain type, Tuple type, Func type

4.2.1 Scalar Type

The scalar type is used define the primitive data type. Different types of Scalar values are Int32, Int64, Float32, Float64 and Bool. The mapping of VTypes to different C++ types is

Method Name	Operation performed	BLAS Call	Specialised Version
mmult	Matrix Multiplication	gemm	Yes
vec_mult	Vector-Matrix Multiplication	gemv	No
scal_mult	Scalar-Matrix Multiplication	scal	Yes
vec_add	Array Addition	axpy	Yes
vec_copy	Array Copy	copy	Yes
vec_sub	Array Subtraction	axpy	Yes
scal_add	Scalar-Array Addition	-	Yes
scal_minus	Scalar-Array Subtraction	-	Yes
transpose	Matrix Transpose	-	Yes

Table 4.4 The table gives a list of array operations that are implemented by the runtime library.

shown in Table 4.5.

Scalar Type		Real / Complex	C++ types
Name	S-Expression		
Float32	(float32)	REAL	float
		COMPLEX	float complex
Float64	(float64)	REAL	double
		COMPLEX	double complex
Int32	(int32)	REAL	int
		COMPLEX	Not Supported
Int64	(int64)	REAL	long
		COMPLEX	Not Supported
Bool	(bool)	REAL	bool
		COMPLEX	Not Supported

Table 4.5 VType to C++ type mapping. The tables shows the different C++ will be mapped to from the VTypes.

4.2.2 Array Types

Array Types are used to define array variables in VRIR. Expressions whose result is an array are also represented by an array type. Variables which are array types are mapped to VrArrays. There are different VrArrays for different data types. The different VrArray types and the VRIR types from which they are mapped are given in Table 4.6.

Array Type			VrArray
Name	Real / Complex	S-expression	
Float64 Array	REAL	(arraytype :ndims :layout (float64 :ctype complex))	VrArrayF64
	COMPLEX	(arraytype :ndims :layout (float64 :ctype complex))	VrArrayCF64
Float32 Array	REAL	(arraytype :ndims :layout (float32 ctype: real))	VrArrayF32
	COMPLEX	(arraytype :ndims :layout (float32,ctype: complex))	VrArrayCF32
Int32 Array	REAL	(arraytype :ndims :layout (int32 ctype: real))	VrArrayI32
Int64 Array	REAL	(arraytype :ndims :layout (int32 ctype: real))	VrArrayI64
Bool Array	REAL	(arraytype :ndims :layout (int32 ctype: real))	VrArrayB

Table 4.6 ArrayType map. The table shows the VrArray types the ArrayTypes in VRIR are mapped to.

4.2.3 Void Type

The void type is used in most cases inside a Func Type to convey the absence of either input or output parameters. The void type is mapped to a simple 'void' in C++.

4.2.4 Tuple Type

Tuple types are used to define data structures which can have data of different types. While generating C++ code, the tuple types are used to generate structs that are in turn used to support data structures containing heterogeneous data.

4.2.5 Domain Type

Domain types are used inside Domain expressions that are described in Subsection 4.6.4. Domain types contain a list of VTypes which represent the VTypes of the iteration variables of a loop.

4.2.6 Func Type

As mentioned in Subsection 3.1.4 func types are associated with function definitions and function handles. They contain information about the types of the input and out parameters of the function. The function types are used for generating function definition. More information about how function definitions are generated in given in section 4.4.

4.3 Modules

Modules are top-level constructs in VRIR. They contain one or more functions which have to be compiled to C++. A module also has an attribute called indexing which defines the indexing scheme. The indexing scheme can either be 0 or 1 indexing. More details about how the indexing attribute is used can be found in section 4.7.1.

4.4 Functions

Functions in VRIR are compiled to separate functions in C++. As mentioned in 3.4, the function node in VRIR has multiple children all of which are required to generate the C++ code for the function. The list of children node of the function node and their role in the code generation process is as follows :

- Name : A C++ function of the same name is generated.
- Arglist : Contains a list of the ids of the input arguments referring to their Symbol table entries. They are used to fetch the names of the input arguments from the Symbol table.
- Func type : The Func type is used for generating the input argument types and the return type.
- Body : It refers to the function body and is converted to C++ statements.

The table 4.7 depicts how a function node in VRIR is converted to a C++ function. The function babai returns an array of type VrArrayF64 and takes as input two arrays of type

VrArrayF64. The return type of the function is determined using the outtype node inside the funcType node. The names and types of the input parameters are determined using the arglist. The ids in the arglist are used to look up the names and types of the input arguments.

Func Type	Generated Function
(function babai (functype (intypes (arraytype :layout colmajor :ndims 2(float64 :ctype 0)) (arraytype :layout colmajor :ndims 2(float64 :ctype 0))) (outtypes (arraytype :layout colmajor :ndims 2(float64 :ctype 0)))) (arglist (arg :id0)(arg :id1)) (body ...)	VrArrayPtrF64 babai (VrArrayPtrF64 R ,VrArrayPtrF64 y)

Table 4.7 The table shows an example func type and the equivalent function signature that was generated using the func type.

4.4.1 Return types in VRIR

C++ only permits single return types. On the other hand, VRIR supports multiple return types. In order to bridge this difference in semantics, we generate a struct definition whose fields are of the same types as the return types. A parameterized constructor is also provided to assign the different variables that need to be returned, to the member fields of the struct. The structure definition and the function definition that returns the structure is shown in Listing 4.3. The struct name is generated using the name of the function. The format of a struct for multiple returns is, struct_<function name>_ret. This allows the calling function to determine the name of the struct while declaring a variable. Listing 4.3 gives a struct definition which has three fields, two for scalar doubles and one for a double array. The Struct also contains a constructor which takes the values of the three fields as input. The constructor is used in the return statement of the function with multiple returns.

```
1  //Structure definition
2  typedef struct struct_adapt_ret {
3     VrArrayPtrF64 ret_data0;
4     double ret_data1;
5     double ret_data2;
6
7     struct_adapt_ret(VrArrayPtrF64 ret_data0,double ret_data1,double ret_data2)
       :ret_data0(ret_data0),ret_data1(ret_data1),ret_data2(ret_data2)
8     {
9     }
10
11 }struct_adapt_ret;
12
13 //Function declaration
14 struct_adapt_ret adapt(double a,double b,double sz_guess,double tol);
```

Listing 4.3 Generated structure to handle multiple returns.

4.5 Statements

VRIR supports various statements such as assignments, for-loops, while-loops etc. Most of the statements are directly supported in C++. The assignment, for and return statements have special cases which need to be supported. VRIR also supports the parallel for statement. This statement is compiled to an equivalent for loop with OpenMP pragmas in C++.

4.5.1 Assignment Statement

While generating C++ code for the assignment statement, we had to take into account different variations of the statement as well cases requiring generation of additional code. Different variations include statements with an array slice operation on the left hand side, statements containing function calls on the right hand side which have multiple returns etc.

Simple Assignment Statements

Simple assignment statements are used when the left hand side is a name expression or an index expression without the array slice operator. The number of expressions on the left

hand side can not be more than one. An example of a simple assignment statement is given in Table 4.8. The example contains an assignment to a scalar double variable temp. The right hand side is a simple index expression with a single index i. The index is subtracted by one because the array was originally one-indexed in the source language. Table 4.8 gives a complete code of the assignment statement. Subsequent examples of will have code with parts that are not relevant to replaced with a statement inside chevrons which describes that particular part of the code.

VRIR	Generated C++ code
(assignstmt (lhs (name :id 5 (float64 :ctype 0))) (rhs (index :arrayid 0 :copyslice %0 (arraytype :layout colmajor :ndims 2 (float64 :ctype 0)) (indices (index :boundscheck %1 :negative %0 (name :id 4 (int64 :ctype 0)))))))	temp = VR_GET_DATA_F64(A) [(i - 1)];

Table 4.8 The table shows an example of the simple assignment statement in VRIR and the equivalent C++ code that is generated from it.

Assignment Statements with Array Slice set

In assignment statements which fall under this category, the left hand side expression is an index expression with at least one slice index. More information about slice indices can be found in Subsection 4.7.2. The right hand side can be any expression. The array slice set operation allows a region of the array to be assigned values. Since C++ does not support array slicing, we have provide a function in the runtime library which implements it. During code generation, the assignment statement is compiled to the function implementing array slice set. The parameters to this call are the array variable of index expression on the left

hand side, the right hand side expression and the set of indices which define the region of the array to which the values have to be assigned. The indices are converted to VrIndex structs. More information on VrIndex can be found Subsection 4.7.2. Table 4.9 shows a VRIR representation with a slice operation on the left hand side and the equivalent C++ code that is generated. The table gives an example of a slice operation with a single index. The slice of the array rrk starting from (k+1) and ending at n is assigned the values of the right hand side expression. The third parameter of VrIndex gives the step value for the range. In this case the step value is one and hence every element from (k+1) to n is considered.

VRIR	C++ backend
(assignstmt (lhs (index :arrayid 12 :copyslice %0 (arraytype :layout colmajor :ndims 2 (float64 :ctype 0)) (indices (index :boundscheck %1 :negative %0 (range :exclude %0 (start (plus (int64 :ctype 0) (lhs (name :id 13 (int64 :ctype 0))) (rhs (realconst :ival 1 (int64 :ctype 0))))) (stop (name :id 8 (float64 :ctype 0)))))))) (rhs <RHS Expression>)))	rrk.setArraySliceSpec (<RHS Expression>, VrIndex((k + 1),n,1));

Table 4.9 Table shows VRIR with array slicing on the LHS and the equivalent C++ code that is generated.

Assignment statements that can be optimised for redundant memory allocations

VRIR	C++ backend
(assignstmt (lhs (name :id 5 (arraytype :layout colmajor :ndims 2 (float64 :ctype 0)))) (rhs (libcall :libfunc mmult (arraytype :layout colmajor :ndims 2 (float64 :ctype 0)) (args (name :id 5 (arraytype :layout colmajor :ndims 2 (float64 :ctype 0))) (name :id 5 (arraytype :layout colmajor :ndims 2 (float64 :ctype 0)))))))	BlasDouble::mmult (CblasColMajor,CblasNoTrans ,CblasNoTrans ,B,B, &B);

Table 4.10 Table shows VRIR with array operations on the LHS and the equivalent C++ code that is generated and optimised.

One of the contributions of the thesis was an optimisation where redundant memory allocations during operations on arrays were removed. This optimisation is implemented by passing the array to which the result is assigned to, as a parameter to the function implementing the array operation. This array is the left hand side expression of the assignment statement whereas the right hand side expression is the function call. More information on the optimisation can be found in Section 6.3. Table 4.10 gives an example of a VRIR representation which can potentially be optimised and the generated C++ code. The example in the table is a call to the matrix multiplication function which is defined inside the runtime library. The call takes as input three BLAS specific parameters, two double arrays and a reference to another double array. The two double arrays are the ones present on the RHS on which the matrix multiplication is performed and third array whose reference is passed

to the function is the array to which the output of the matrix multiplication operation is assigned to. Since the LHS is passed as input, no assignment operator(=) is used.

Assignment Statements with multiple LHS expressions

VRIR	C++ backend

(assignstmt
(lhs
(tuple
(tupletype
(float64 :ctype 0)
(arraytype :layout colmajor:ndims 2
(float64 :ctype 0)
)
(arraytype :layout colmajor:ndims 2
(float64 :ctype 0))
(arraytype :layout colmajor:ndims 2
(float64 :ctype 0)
)
)
(elems
(name :id 3
(float64 :ctype 0)
)
(name :id 8
(arraytype :layout colmajor:ndims 2
(float64 :ctype 0)
)
)
(name :id 4
(arraytype :layout colmajor:ndims 2
(float64 :ctype 0)))
(name :id 9
(arraytype :layout colmajor:ndims 2
(float64 :ctype 0)
)
)
)
)
)
(rhs
<RHS Expression>
)
)
	struct_spqr_ret var_spqr1 =
	<rhsExpr>
	nr = var_spqr1.ret_data0;
	S = var_spqr1.ret_data1;
	rx = var_spqr1.ret_data2;
	rn = var_spqr1.ret_data3;

Table 4.11 Table shows VRIR with multiple expressions on the LHS and the equivalent C++ code that is generated.

Assignment statements can have multiple expressions on the LHS when the RHS expression call to a function with multiple returns. As mentioned in section 4.4, functions with multiple returns are handled by returning a struct containing the return values. In the

assignment statement the expression on the LHS are replaced by a struct variable and additional code is generated to assign the values in the struct to the LHS expressions. Table 4.11 shows the C++ code that is generated from VRIR. The example shown in the table shows a variable that is of the structure type struct_spqr_ret which is assigned the value of the RHS expression. The different variables are then assigned the values in the different fields of the structures in the subsequent statements.

4.5.2 For Statement

The For statement node in VRIR is compiled to a for loop is C++. The domain expression node of the for statement is used to determine the ranges over which the for loop iterates. If there are multiple ranges in the domain node, the for statement node is compiled into multiple nested loops. The names of the loop variables is determined by fetching their IDs from the itervar node and using their IDs to look up their names in the symbol table. An example of the generated in given in table 4.12. The example shows a simple for statement which is converted into the standard C++ for loop. The loop variable is h, which has an initial value of 1 and a final value of k. The loop iterates from a smaller initial to a larger final value and with each iteration, the value of the loop variable is incremented by one.

Determining loop direction

While generating code for the for statement, the direction of the loop can be determined by the start, stop and step values. Table 4.13 shows the directions of a for loop for different values of start,stop and step. The loop direction can only be determined if the value of the step value is known during compilation. In order to determine the loop direction we check whether the step expression is a constant expression or a negate expression with a constant expression as its child.

Generating the loop vector

If the direction of the loop cannot be determined at compile time, we declare a vector in the generated code. All possible values of the loop variable will be inserted into the vector at run time. The generated loop iterates over the vector and the loop variable is assigned

VRIR	C++ backend

(forstmt
(itervars
(sym :id 7)
)
(loopdomain
(domain
(domaintype :ndims 1
(int64 :ctype 0)
)
(range :exclude %0
(start
(realconst :ival 1
(int64 :ctype 0)
)
)
(stop
(name :id 3
(float64 :ctype 0)
)
)
)
)
)
(body
<Loop Body>
)
)
	for(h=1;h<= k;h=h+static_cast<long>(1)) {
	<Loop Body>
	}

Table 4.12 The table shows a for statement node in VRIR and its equivalent C++ code

Start and Stop values	Step Value	Loop Direction
Stop >Start	Negative	Empty Loop
	Positive	Increment
Stop <Start	Negative	Decrement
	Positive	Empty Loop

Table 4.13 Table shows the direction of a for loop for various start, stop and step values

consecutive values of the vector inside the loop body. Listing 4.4 gives an example of the generated C++ code for a loop when the direction of the loop cannot be determined at compile time. As we can see in line 1, a vector is initialised through the function call getIterArr. The start, stop and step expressions, namely m, nn and istep are passed as parameters to the functions. A for loop iterating over the vector is generated as can be seen in line 2. Finally, consecutive values of the vector are assigned to the loop variable i. This can be seen on line 3.

```
std::vector<long> vrTempVec0 = getIterArr<long>(m,nn,istep);
for( long vrTempIter0 = 0 ; vrTempIter0 < vrTempVec0.size(); vrTempIter0++ ){
    i=vrTempVec0[vrTempIter0];
    <Loop Body>
  }
```

Listing 4.4 The listing gives an example of generated C++ code when the loop direction cannot be determined

Determining inclusion of the Stop value

For loop excluding stop value	`for(h=1; h< k; h=h+static_cast<long>(1)) {`
	`  <Loop Body>`
	`}`

For loop including stop value	`for(h=1; h<= k; h=h+static_cast<long>(1)) {`
	`  <Loop Body>`
	`}`

Table 4.14 Table shows a C++ for loop with the exclude flag set to 0 and 1.

The loop domain node of the for statement gives the start, stop and step expressions for each range. These expressions are used to generate the initialisation, condition and increment statements of the C++ for loop. We have to determine whether the range is inclusive of the stop value. This is done using the exclude flag of the range expression. If the flag is set to '%1', the value is excluded whereas it is included if the flag is set to '%0'. In case of an excluded stop value, the '<' or the '>' operator is used in the condition statement and the '<=' or the '>=' operator is used in case of an included stop value. Table 4.14 gives an example the generated for loops with and without including the stop value.

4.5.3 Return Statement

For return statements with single return variable, a simple return statement is generated. The expressions inside the return statement are replaced with their Ids inside the symbol table. If the expressions are not name expressions, they are assigned to a temporary variables which are then returned. Table 4.15 gives an example of a simple return statement. In the example, the return statement returns a variable called cap. Since C++ does not

VRIR	Generated C++
(returnstmt (exprs (name :id 7 (float64 :ctype 0))))	return cap;

Table 4.15 The table shows a return statement with a single return value and its equivalent C++

support return statements with multiple variables, we return a struct instead. The values of the variables are passed as parameters to the struct's constructor. Table 4.16 gives an example of the return statement with multiple returns values. The example returns struct of type struct_spqr_ret. The struct object is created by means of a constructor that takes as parameters the variables that need to be returned.

VRIR	Generated C++
(returnstmt (exprs (name :id 3 (float64 :ctype 0)) (name :id 4 (arraytype :layout colmajor :ndims 2 (float64 :ctype 0))) (name :id 5 (arraytype :layout colmajor :ndims 2 (float64 :ctype 0))) (name :id 6 (arraytype :layout colmajor :ndims 2 (float64 :ctype 0)))))	return struct_spqr_ret (ncols,R,colx,norms);

Table 4.16 The table shows a return statement with multiple returns and its equivalent C++

4.5.4 If Statement

The if statement is compiled to a condition statement, an if block and an else block if it exists. Table 4.17 gives an example of the if statement.

VRIR	Generated C++

(ifstmt
(test
<Test Condition>
)
(if
<If Block>
)
(else
<Else block>
)
)
	if(Test condition) {
	<If Block>
	} else {
	<Else Block>
	}

Table 4.17 The table shows an example of a if statement in VRIR and its equivalent C++ code.

4.5.5 Break and Continue Statement

The break and continue statements of VRIR are compiled to the break and continue statements in C++.

4.5.6 While Statement

The while statement is compiled to a while loop in C++. The test node of the statement is used to generate the while condition. Statements inside the body node are compiled to the statements inside the loop body. Table 4.18 gives an example of the while statement.

VRIR	Generated C++

(while
(test <While Condition>)
(body
<Loop Body>
)
)
	while(condition) {
	<Loop Body>
	}

Table 4.18 The table shows an example of a while statement in VRIR and its equivalent C++ while loop

4.5.7 Parallel For Statement

A parallel for loop is compiled to a for loop in C++ with an OpenMP pragma inserted before the loop. The shared variables node of the parallel for statement contains IDs of the variables that are shared and are added to the shared option of OpenMP. The list of shared

variables are provided by the language specific frontend. Private variables are defined by generating a list of variables defined inside the loop and removing the ones that are present in the shared variable list. Table 4.19 gives an example of the parallel for statement. The OpenMP pragma gives a list of shared variables, namely, A,B and c and a list of private variables.

VRIR	Generated C++

(pfor
(itervars
(sym :id 6
)
)
(loopdomain
(domain (domaintype :ndims 1
(int64 :ctype 0))
(range :exclude %0
(start
(realconst :ival 1
(int64 :ctype 0)
)
)
(stop
(name :id 4
(float64 :ctype 0)
)
)
)
)
)
(shared 3 4 5)
)
	#pragma omp parallel for\
	shared(A,B,c)
	for(i = 0;
	i < m;
	i++) {
	<Loop Body>
	}

Table 4.19 The table shows an example of a parallel for statement in VRIR and its equivalent C++ for loop with OpenMP

4.6 Expressions

Most expressions VRIR can be compiled to equivalent expressions in C++. Expressions such as index expressions have special cases which need to be considered. The following subsections explain the compilation of the different VRIR expressions.

4.6.1 Operators

Many binary and unary expressions in VRIR can be classified as arithmetic operators. These include binary expressions such as plus, minus, mult, div and the negate expression. These expressions only support scalar operands. Hence generating C++ code these operators is straightforward. They are mapped to the operators directly supported by C++. Thus plus is mapped to the '+' operator in C++, minus is mapped to '-'. The complete list is given in Table 4.20.

Operations on arrays, on the other hand, are mapped to the LibCall expression in VRIR.

VRIR operators	C++ Operators
plus	+
minus	-
mult	*
div	/
and	&&
or	||
lt	<
leq	<=
gt	>
geq	>=
eq	==
neq	!=

Table 4.20 VRIR operators to C++ operators Mapping. The Table shows the C++ operators to which the VRIR operators are mapped.

Since C++ does not support operators for arrays we implemented functions to support these operations on arrays. These functions are housed inside the language-specific runtime library. Where ever possible these functions make calls to BLAS functions for enhanced performance. In case of the MATLAB runtime, we use the Intel Math Kernel Library[Cor] or MKL implementation of BLAS and in case of Python, we use the OpenBLAS[ZX] implementation. Table 4.21 gives the list of function calls generated in C++ for array operations.

VRIR Lib Call	Operand 1	Operand 2	C++ function
Matrix Multiplication	Array	Array	mmult
	Array	Scalar	scal_mult
Elementwise Multiplication	Array	Array	vec_mult
	Array	Scalar	scal_mult
Matrix Left Division	Array	Array	mat_ldiv
	Array	Scalar	scal_div
Matrix Right Division	Array	Array	mat_rdiv
	Array	Scalar	scal_div
Elementwise Division	Array	Array	elem_div
	Array	Scalar	scal_div
Array Addition	Array	Array	vec_add
	Array	Scalar	scal_add
Array Subtraction	Array	Array	vec_sub
	Array	Scalar	scal_minus
Array Copy	Array	Array	vec_copy
Matrix Transpose	Array	-	transpose

Table 4.21 The table shows the different C++ functions array operators are mapped to.

4.6.2 Name Expressions

The name expressions in VRIR denote variables. The 'Id' attribute of the name expressions is used to fetch the symbol string from the symbol table. All name expressions that are not passed as parameters to the function are declared at the start of the function body. Table 4.22 gives an example of name expressions. In the example, the name expression that has an id of 2 and is of type int64 is converted to a variable A.

VRIR	Generated C++
`(name :id 2 (int64 :ctype 0) )`	`A`

Table 4.22 The table shows an example of a name expression in VRIR and its equivalent C++ symbol

4.6.3 Function call expressions

Function call expressions in VRIR are used to describe calls to functions that are not defined by the library call expression or the alloc expression. A function call expression may have zero or more arguments. Arguments can be passed by reference or a copy of the arguments could be passed to the function. We define certain functions as builtins. These are functions that we support through the runtime library. Arguments to builtins are always passed by reference. The Table 4.23 gives an example of the function call expression. The example is a call to the function mean. As explained in Section 4.1 since the function is a builtin and it returns a scalar value and, the function mean_scalar is generated.

VRIR	Generated C++

`(fncall :fnname mean`
`  (float64 :ctype 0)`
`  (args`
`  (name :id 22`
`  (arraytype :layout colmajor :ndims 2`
`  (float64 :ctype 0)`
`  )`
`  )`
`  )`
`)`
	`vr_temp37 = mean_scalar(frx);`

Table 4.23 The table shows an example of a function call expression in VRIR and its equivalent C++ expression

4.6.4 Domain Expression

Domain expressions are used inside for statements to define the ranges of the for loops. A domain expressions can have one or more ranges. All domain expressions are of the domain type. Table 4.12 gives an example of how domain expressions inside a for statement are used to generate a for loop in C++. Domain expressions are always found inside for statements.

4.6.5 Constant Expressions

Constant Expressions hold constant values in VRIR. They are compiled to constants inside C++. The type of a constant expressions is defined by the vtype node. A real constant can

either have an ‘ival’ or a ‘dval’ attribute which defines an integer value or a floating point value respectively. Table 4.24 gives an example of constant expressions.

AttributeType	VRIR	Generated C++
dval	`(realconst :dval 2.3e-12(float64 :ctype 0))`	`2.3e-12`
dval	`(realconst :dval 2(float64 :ctype 0))`	`2.0f`
ival	`(realconst :ival 2(int64 :ctype 0))`	`2`

Table 4.24 The table shows an example of a constant expression in VRIR and its equivalent C++ constant

4.6.6 Alloc Expression

Alloc expressions are used to define functions which allocate memory and initialise it. The expression defines three types of functions zeros, ones and empty each of which are compiled to function calls in the run time library. Table 4.25 gives an example of an alloc expression for the zeros function. The generated C++ has an additional parameter to define the number of input parameters. This is because the zeros function call in the runtime library variable arguments.

VRIR	Generated C++

`(alloc :func zeros`
`  ( arraytype :layout colmajor :ndims 2`
`  (float64 :ctype 0)`
`  )`
`  (args`
`  (name :id 2`
`  (float64 :ctype 0)`
`  )`
`  (name :id 4`
`  (float64 :ctype 0)`
`  )`
`  )`
`)`
	`zeros(2,m,k);`

Table 4.25 The table shows an example of an alloc expression in VRIR and its equivalent C++ symbol

4.6.7 Dim Expression

Dim Expressions are used to fetch the size of the specific argument of an array. Dim Expressions are compiled to a call to the size function in the runtime library. Table 4.26 gives example of a dim expression. The attribute arrayid gives the id of the array in the symbol table. The attribute dimid gives the dimension whose size is requested. The C++ code generated is a function call to size.

VRIR	Generated C++
`(dim :arrayid 0 :dimid 0 (int64 ) )`	`size(A,0);`

Table 4.26 The table shows an example of an dim expression in VRIR and its equivalent C++ symbol

4.6.8 Tuple Expression

Tuple expressions are used as for containers for heterogeneous data in VRIR. Return values of function calls with multiple returns are assigned to a tuple expressions. The Tuple expressions are also used for MATLAB's cell arrays and matrix expressions and Python's tuples. The Table 4.27 gives an example of the tuple expression. The tuple contains 2 elements, one is a scalar of type float64 and the other is an array. A struct is generated which has two member fields of a scalar and a array type.

4.6.9 Cast Expressions

Cast Expressions are used to cast an expressions of a certain type to a different type. We assume that the cast is valid and do not add any code to check its validity. Cast Expressions are compiled to a static_cast in C++.

VRIR	Generated C++

(tuple
  (tupletype
  (float64 :ctype 0)
  (arraytype :layout colmajor :ndims 2
  (float64 :ctype 0)
)
)
  (elems
  (name :id 3
  (float64 :ctype 0)
)
  (name :id 8
  (arraytype :layout colmajor :ndims 2
  (float64 :ctype 0)
)
)
)
)
	struct_spqr_ret var_spqr0 =
	  <LHS Expression>

Table 4.27 The table shows an example of an tuple expression in VRIR and its equivalent C++ symbol

4.7 Index Expressions

Index expressions in VRIR used to define indexing on arrays. Index expressions have one or more indices. The number of indices is not dependent on the number of dimensions of the array. We classify indexing on arrays into two types, basic indexing and advanced indexing. Flags such as boundscheck and negative define whether boundscheck code needs to be generated for the expressions and whether the index expression supports negative indexing respectively.

4.7.1 Basic Indexing

Indexing is defined as basic if all the indices are scalars. The indices can also have negative values. The generated code look similar to an array index in C++. However, since VrArray contains a single dimensional pointer to the array data, we have to reduce multiple index values to a single index value during code generation.

VRIR	Generated C++
(index :arrayid 5 :copyslice %0 (float64 :ctype 0) (indices (index :boundscheck %1 :negative %0 (name :id 8 (int64 :ctype 0))) (index :boundscheck %1 :negative %0 (name :id 6 (int64 :ctype 0)))))	vr_temp9 = VR_GET_DATA_F64(c) [(i - 1) + VR_GET_DIMS_F64(c)[0]*((j - 1))];

Table 4.28 The table shows an example of an index expression in VRIR with basic indexing and its equivalent C++ symbol

Generating a single index value from multiple indices

Generating a single index value is dependent on the array layout. We support both row and column major array layouts and hence support generating single index value generation for both. In case of a row major layout, the last value is contiguous and hence single index value is given by[1],

$$n_d + N_d \cdot (n_{d-1} + N_{d-1} \cdot (n_{d-2} + N_{d-2} \cdot (\cdots + N_2 n_1) \cdots))) = \sum_{k=1}^{d} \left(\prod_{\ell=k+1}^{d} N_\ell \right) n_k \quad (4.1)$$

where, n_i is the i^{th} index and N_i is the i^{th} dimension of the array.
And for a column major layout, the first value is contiguous and hence the single index value is given by,

$$n_1 + N_1 \cdot (n_2 + N_2 \cdot (n_3 + N_3 \cdot (\cdots + N_{d-1} n_d) \cdots))) = \sum_{k=1}^{d} \left(\prod_{\ell=1}^{k-1} N_\ell \right) n_k \quad (4.2)$$

where, n_i is the i^{th} index and N_i is the i^{th} dimension of the array.
Table 4.28 gives an example of an index expression and its equivalent generated C++ code. The array layout is column major in the case of this example.

[1]Source:`http://en.wikipedia.org/wiki/Row-major_order`

Negative Indexing

Languages such as Python support negative indices. The index refers to an offset from the end of the array dimension. Since C++ does not support negative indexing, we replace the indexing scheme mentioned in Subsection 4.7.1 with a call to the function getIndexVal. Since it is difficult to determine at compile time if all the indices are non-negative, we make a pessimistic assumption that atleast one of the indices will be negative and generate a function call if the negative flag in the index expression is set to 1. Table 4.29 gives an example of an index expression with negative indexing. The function getIndexVal generates the appropriate index value. The first parameter is a describes the array layout. The value 0 means that the array layout is row major, the value 1 means column major and 2 means strided.

VRIR	Generated C++
(index :arrayid 0 (float64) (indices (index :boundscheck %1 :negative %1 (name :id 5 (int64))) (index :boundscheck %1 :negative %1 (realconst :ival 0 (int32)))))	VR_GET_DATA_F64(a)[getIndexVal_spec<VrArrayPtrF64> (0,a, i,1)]

Table 4.29 The table shows an example of an index expression in VRIR with negative indexing and its equivalent C++ symbol

4.7.2 Advanced Indexing

We define cases where the array indices are non-scalar as advanced indexing. The indices can either be arrays or ranges. The index expression is compiled to a function call which returns an appropriate value for the given input indices. The function takes as input, arguments of type VrIndex, a struct defined in the runtime library.

VrIndex

The VrIndex struct is shown in Listing 4.5. The structure contains two boolean flags m_isRange and m_isArray to differentiate which are used to determine whether the index is a range or an array respectively. If both flags are set to false, the index is a constant value. The constant value is stored in the variable const_val. The range is stored as an array of size 3. The elements of the array are the start, stop and step values, in order. The array value is stored in the variable arr.

```
struct VrIndex{
  bool m_isRange;
  bool m_isArray;
  VrArrayF64 arr;
  union Val{
    dim_type const_val;
    dim_type range_val[3];
  }m_val;
  VrIndex(dim_type const_val);
  VrIndex(dim_type start,dim_type stop,dim_type step);
  VrIndex(VrArrayF64 A);
  VrIndex();
};
```

Listing 4.5 VrIndex Structure

Array Slicing

Array slicing operations extract certain elements of an array. We define two types of array operations, the array slice get and the array slice set. An array slicing operation is performed if one or more of the indices of an index expression in VRIR is a range expression. The range defines elements to be extracted. Index expressions can contain a combination of range expressions and other expressions having scalar as well as array types. Hence each expression is converted to a VrIndex struct. Array slicing operations are implemented as struct methods of VrArrays. The method for array slice get is called sliceArray and that for array slice set is setSliceArray. Specialised versions of the methods for one, two and three indices are also implemented. The specialised versions for array slice get and set are

sliceArraySpec and setSliceArraySpec respectively. Table 4.30 gives an example of an array slice get operation. The generated C++ code is a function call to the specialised version for two indices, sliceArraySpec. The first parameter is a simple scalar index k, where as the second parameter is a range from k+1 to n. Both parameters are converted to VrIndex structs through constructors.

VRIR	Generated C++

`(index :arrayid 0 :copyslice %0`
`  ( arraytype :layout colmajor :ndims 2`
`  (float64 :ctype 0))`
`  (indices`
`  (index :boundscheck %1 :negative %0`
`  (name :id 4`
`  (float64 :ctype 0)`
`  )`
`  )`
`  (index :boundscheck %1 :negative %0`
`  (range :exclude %0`
`  (start`
`  (plus`
`  (float64 :ctype 0)`
`  (lhs`
`  (name :id 4`
`  (float64 :ctype 0)`
`  )`
`  )`
`  (rhs`
`  (realconst :dval 1`
`  (float64 :ctype 0)`
`  )`
`  )`
`  )`
`  )`
`  (stop`
`  (name :id 3`
`  (float64 :ctype 0)`
`  )`
`  )`
`  )`
`  )`
`  )`
`)`
	`R.sliceArraySpec(VrIndex(k)`
	`  ,VrIndex((k + 1),n,1))`

Table 4.30 The table shows an example of an index expression in VRIR that is converted to an array slicing function call in C++

Chapter 5
Glue Code Generation

The Velo**C**ty compiler generates C++ code for functions identified as computationally intensive by the user. The rest of the code is not compiled. Thus, since the computationally intensive functions and the remaining part of the program are in two different programming languages, namely C++ and the source language, an interface between the two code sections is required. Most high-level languages provide an API to interface with C/C++. PyVrir generates the required interface or glue code for Python. However, no glue code generator exists for MATLAB. Hence, along with generating VRIR, we also generate C++ code required to interface MATLAB programs with the generated functions. The MATLAB MEX API is used for the interface.

5.1 Generating code for including header files

Header files are required for the following reasons.

- Declarations of MEX functions.
- Declaration of functions in the runtime library.
- Declaration of OpenMP functions.

The header files are included using the "include" preprocessor directive. Listing 5.1 gives an example of the header files that are generated. The header file "mex.h" provides the set of

```
#include<mex.h>
#include"matrix_ops.hpp"
#include"library_ops.hpp"
#include"matmul_pImpl.hpp"
#include<omp.h>
```

Listing 5.1 Example of header files in glue code

declarations for the MEX API functions. The header files matrix_ops.hpp, library_ops.hpp contain class and function declarations of our runtime library. Function declarations of the generated code are provided by matmul_pImpl.hpp. The file omp.h is provided for OpenMP functions and directives.

5.2 Generating mexFunction

The entry point for any shared library that can be called from MATLAB is called mexFunction. Listing 5.2 gives an example of the mexFunction. The function returns void. It takes four input arguments. The first argument nlhs defines the number of output parameters of the function and the second argument is an array of output parameters. The output parameters are of type mxArray which is a MATLAB specific array representation.

```
void mexFunction(int nlhs, mxArray *plhs[],
    int nrhs,const mxArray *prhs[])
```

Listing 5.2 The entry point function for the MEX API

5.2.1 Generating VrArrays from mxArrays

The arrays in the generated functions are represented as VrArrays. VrArrays are a VeloCty specific representation of arrays. VrArrays contain the array data as well as metadata such as the number of dimensions and the dimensions themselves. More information on VrArrays can be found in Subsection 4.1.1. VrArrays were used in place of the language specific representation because accessing data and metadata of mxArrays was expensive. Since the

arrays are passed as mxArrays from MATLAB and the arrays are represented as VrArrays in the generated code, the glue code has to convert the mxArrays passed as input from MATLAB to VrArrays before they can be passed to the generated functions.

We have implemented methods which convert mxArrays to VrArrays. There is a separate method for each VrArray type. Listing 5.3 gives an example of the function used to convert mxArrays to VrArrays. The function is called getVrArrayF64 which takes a pointer to a mxArray as input and returns a VrArrayF64, an array of doubles. Table 5.1 gives a list of all the functions for converting VrArrays to different mxArrays.

```
1 VrArrayF64 y = getVrArrayF64(rhs[1]);
```

Listing 5.3 Converting mxArrays to VrArrays

Function	Description
getVrArrayF64	Returns an array of doubles
getVrArrayF32	Returns an array of floats
getVrArrayCF64	Returns an array of complex doubles
getVrArrayCF32	Returns an array of complex floats
getVrArrayI32	Returns an array of 32 bit integers
getVrArrayI64	Returns an array of 64 bit integers

Table 5.1 List of functions used to convert mxArrays to VrArrays.

There is an additional overhead while converting from complex mxArrays to complex VrArrays because the representation of the data in the two array types is different. mxArrays store the real and imaginary data as separate arrays. On the other hand, in VrArrays, the real and imaginary data is interleaved. Thus, for every element of the array, the real value is immediately followed by the imaginary value.

Scalar values are also passed as mxArrays by MATLAB where as they are represented using the C++ primitive types. The glue code also converts the mxArrays to scalar types when required. Listing 5.4 gives an example of the conversion. The example shows a mxArray pointer rhs[5] being converted to a scalar value inputData5. The MEX function, mxGetScalar returns a scalar double value. A cast is required for all types other than double.

```
1  double inputData5 = static_cast<double>(mxGetScalar(rhs[5]));
```

Listing 5.4 Converting mxArrays to scalars

5.2.2 Function Call

Once all the input parameters are converted to VrArrays or C++ primitive types, we make a call to the generated entry point function. The output of the function is stored in either a VrArray or a scalar variable if the function returns a single variable. Listing 5.5 gives an example of the generated function call. The listing shows a call to the function babai which returns a VrArrayF64 which is assigned to the variable retVal.

```
1  VrArrayF64 retVal = babai(R,y);
```

Listing 5.5 Call to generated function

The generated function can also return multiple variables of different types. In this case the generated function packages the variables into a struct and returns the struct. More information about multiple returns can be found in Subsection 4.4.1. Listing 5.6 gives an example of a call to a function with multiple returns. The function name is nb1d which takes 7 inputs and returns a struct of type struct_nbody1d_ret retVal.

```
1  struct_nbody1d_ret retVal = nbody1d(inputData0,inputData1,
2                inputData2,inputData3,
3                inputData4,inputData5,inputData6);
```

Listing 5.6 Call to generated function

5.2.3 Converting to mxArrays

The output of the generated function has to be returned to MATLAB. The output can consist of a single or multiple variables. The output can be returned via the array plhs. As mentioned earlier, plhs is an array of mxArray pointers. Hence the output of the generated function, which is either a VrArray or a C++ primitive type has to be converted to

mxArrays and stored as successive plhs elements. We use MEX API functions to do the same.

In case of VrArrays, we first have to create an array of the required size. We do this using the function mxCreateNumericArray. This function takes as input, the number of dimensions, an array of dimension sizes, the data type of the array and its complexity. We use the ndims and dims fields from the VrArray to specify the number of dimensions and the sizes of each dimensions. Once the array is created, we set the mxArray data by passing the pointer to the data inside the VrArray to the the function mxSetData.

For scalar values, we use the MEX function mxCreateNumericMatrix which accepts the row and column sizes as well as the complexity and the type of the array.We set the row and column sizes as one. We then fetch a pointer to the data of the newly created mxArray and set the zeroth element to the scalar value.

If the function returns multiple output values, each data member of the return struct is used to create an mxArray which is then assigned to successive indices of plhs.

Chapter 6
Code Optimisations

The primary goal of the thesis was to ensure correct compilation of code from MATLAB and Python to C++. An additional goal was to improve the performance of the generated code. Initial experiments showed that turning on bounds check slowed down 6 of the 17 benchmarks and 3 of the 9 benchmarks in Python. The geometric mean of the slowdown compared to bounds check turned off was 3.66 for MATLAB and 1.63 for Python. Additionally, while analysing the generated code, we found that array operations being performed inside loops were allocating memory to the same output array for every iteration. We determined that by optimising the code to eliminate bounds checks and unnecessary memory allocations, we gain a significant improvement in performance. In this chapter we first discuss the bounds check implementation followed by the two optimisations, namely elimination of bounds checks and elimination of redundant memory allocations.

6.1 Bounds Checks

Scientific languages like MATLAB and Python support array bounds checks for indexing operations. These checks ensure that the program does not crash abruptly and instead throws an error before exiting. On the other hand, C++ does not implicitly support array bounds checks. Hence we provide bounds checks through the runtime library.

Due to differences in semantics of MATLAB and Python, the bounds check implementations for both languages are different. Hence we provide different implementations for

the two languages. Array growth is also carried out by the bounds check implementation for languages that support it.

However, the API for both language implementations is the same. The entry point function for bounds checks is a templated function called checkBounds. Listing 6.1 gives an example of the bounds check function for the array c. The bounds check functions are called inside conditional blocks which allows the user to turn the checks on or off while compiling the code. The first parameter is the reference to array on which the indexing operation is performed. The second parameter is a boolean flag which is set to true if the boolean operation is on the LHS of an assignment statement. This flag is used to determine if the array should be grown when one or more indices exceed bounds. This check is only used by the MATLAB implementation. The third parameter is the number of indices. This parameter is required since the function accepts variable arguments. The remaining parameters are the indices which are passed as VrIndex structs. Passing indices as VrIndex structs allows the function to handle different index types such as ranges and arrays.

```
1 //Bounds check
2 #ifdef BOUND_CHECK
3   checkBounds<VrArrayPtrF64,double>(&c,false,2,vrIndex(i),vrIndex(j));
4 #endif
```

Listing 6.1 An example of the bounds check function call.

However, the default bounds check function performs poorly due to dynamic memory allocation. The implementation inserts the indices into an array and performs checks while iterating over the array. Using an array simplifies the code for the checks. However, since the number of indices can vary, the array cannot be created at compile time.

In order to improve performance of the bounds checks, we implemented specialised versions of the bounds check function for index operations with one, two and three indices. We also implemented three additional versions for index operations where all indices have numeric values. These specialised functions are called checkBounds_spec. Listing 6.2 shows an example of the specialised version of the bounds check function. The function is specialised for two indices, both of which have numeric values. The first two parameters denote the array and whether the operation is performed on the LHS like in the default

```
1 //Bounds check
2 #ifdef BOUND_CHECK
3   checkBounds_spec<VrArrayPtrF64,double>(&c,false,
4     static_cast<dim_type>(i),static_cast<dim_type>(j));
5 #endif
```

Listing 6.2 An example of the specialised bounds check function call

function. The remaining two parameters are the indices.

6.2 Bounds Check Elimination

Slowdown when the bounds checks was identified to be higher when the checks are performed inside loop bodies. In such cases, the checks are performed for every loop iteration resulting in the slowdown. Consider the example given in Listing 6.3. The example contains an index expression on the array y inside a loop. The index expression consists of a single index (k-1). The loop has a starting value of 1 and a final value of n-1. The value of the loop variable k is incremented by one with every iteration. As we can see, the index is a linear function of the loop variable k. The loop bounds are not modified inside the loop. Moreover, the step value of the loop is a constant and hence it can be inferred that the loop direction is upwards, that is, the loop iterates from a smaller start value to a larger stop value. By replacing the loop variable by the start and stop expressions of the loop, we get the lower and upper bounds of the index respectively. Thus the smallest value of the index inside the loop will be (1-1), that is, 0 and the largest value of the index will be ((n-1) - 1), that is, (n-2). Since we know what the lower and upper bound of the index, we can check whether these values exceed the array size or are less than the lowest index value supported by indexing scheme, outside the loop. If the index is valid, there is no need to perform bounds checks inside the loop. Thus this technique would improve the performance of the program. We use this optimisation technique, on a subset of the indices known as affine indices.

```
1    for(k=1;k<=(n-1);k=k+1)
2        {
3   #ifdef BOUND_CHECK
4           checkBounds_spec<VrArrayPtrF64,double>(&y,false,
5         static_cast<dim_type>(k-1));
6   #endif
7           vr_temp12 = VR_GET_DATA_F64(y)[(k - 1)];
8       }
```

Listing 6.3 Example C++ for loop with array index expressions

6.2.1 Affine indices

A function of one or more variables is considered to be affine if it can be expressed as a sum of constant and constant multiples of the variables. Equation 6.1 gives a mathematical representation of affine functions.

$$f = C_0 + \sum_{i=1}^{n} C_i X_i \tag{6.1}$$

where C_i is the i^{th} constant and X_i is the i^{th} variable.

Affine indices can be defined as array indices which are affine functions of the loop induction variables. Table 6.1 gives examples of affine and non-affine array indices.

Array Index	Affine
A(2*i+1)	Yes
A(i-1)	Yes
A(i*j)	No
A(i*i)	No
A(b(i))	No

Table 6.1 Examples of affine and non-affine indices

6.2.2 Technique

The process of moving the checks outside the loop body can be divided into two parts. Identifying index operations which can be moved outside the loop and generating a if con-

dition for the checks and two versions of the loop body.

Identifying valid index operations

We define valid index operations to have the following properties.

1. All indices should be affine functions of the loop variables.
2. All the loop variables should have loop invariant bounds.

In order to determine whether a check for an index operation can be moved outside the loop body, we check whether individual indices are affine. Indices in VRIR are represented by IndexStructs. We do not consider indices which are ranges or expressions with non-scalar types. For indices with scalar expressions, we recursively traverse the expression until we reach a name or a constant expression or we reach an unsupported expression. Constant expressions are considered affine. In case of name expressions we check whether the expression is a loop invariant or a loop variable. If the expression is a loop variable, we check whether the loop bounds are loop invariant. Apart from constant and name expressions, we also support binary and unary expressions. We only support a specialised case of the mult expression where the LHS and RHS of the expression are either name or constant expressions. If both the LHS and the RHS are name expressions, they should both not be loop variables. We support the same expressions for checking the validity of loop bounds. The set of supported expressions are given in Table 6.2.

Expression Name	Description
plus	Scalar Addition
minus	Scalar Subtraction
name	Variable Name
negate	Unary Minus
const	Constant Value

Table 6.2 List of supported expressions for affine index check

Generating code

To implement the optimisation, the compiler generates an if statement. The if condition contains the checks for the valid index operations. For every index operation, we perform two checks. One for the lower bounds and another for the upper bounds. These checks are performed through functions called checkDimStart for the lower bounds and checkDimStop for the upper bounds. The functions takes as input integer indices. VrIndex structs are not required since indices containing ranges or having non-scalar types are not considered to be valid by the analysis. These functions are implemented inside the runtime library. The specialised functions for one two and three indices are also implemented in the library. Listing 6.4 gives an example of the default and specialised functions. The default functions take the array name, the number of indices and the indices as parameters. The specialised functions named <default function name>_spec take the array name and the indices as parameters. In the example, the functions take 2 indices as input. The loop variables are replaced by the lower and upper bounds of the loops when being passed as arguments to the check functions.We use the loop direction to determine whether the loop variables need to be replaced by the lower bounds for checkDimStart and upper bounds for checkDimStop or vice versa. If loop direction is up, that is the lower bound value is smaller than the upper bound value, the lower bound are used in checkDimStart and upper bound for checkDimStop.

Listing 6.5 gives an example of the if statement generated by the compiler. The example shows a total of 6 functions, three for the lower bounds and three for the upper bounds of three arrays A, B and c. If the checks return true, a checks free version of the code is executed else the default version with checks is turned on is executed.

6.3 Eliminating unnecessary memory allocations

Array operations and array slicing are implemented through functions in the runtime library. The output of these operations is written to a new array created inside the functions. Many times these operations are performed inside loops and the output is assigned to the same array variable. However, runtime memory allocation in expensive. Consider the

```
//Default function
checkDimStart<VrArrayPtrF64>(c,2,1,1)
checkDimStop<VrArrayPtrF64>(c,2,m,n)

//Specialised function
checkDimStart_spec<VrArrayPtrF64>(c,1,1)
checkDimStop_spec<VrArrayPtrF64>(c,m,n)
```

Listing 6.4 An example of the default and specialised function calls for the boundscheck optimisations

```
if(checkDimStart_spec<VrArrayPtrF64>(c,1,1) && checkDimStop_spec<VrArrayPtrF64>(c,m,n) &&
   checkDimStart_spec<VrArrayPtrF64>(B,1,1) && checkDimStop_spec<VrArrayPtrF64>(B,k,n) &&
   checkDimStart_spec<VrArrayPtrF64>(A,1,1) && checkDimStop_spec<VrArrayPtrF64>(A,m,k)) {
     <For Statements without bounds check >
   } else {
     <For Statements with bounds check >
   }
```

Listing 6.5 An example of the if statement generated for the boundscheck optimisations

example in Listing 6.6. The example contains an array operation, scal_minus, which subtracts a scalar vr_temp28 from every element in the array Rx and assigns it to an array that is created inside the function. The output of this operation is assigned to another array drx. Since this function is called inside the loop, a new array would be created on every loop iteration. However output array that is created will always be assigned to drx. The number of memory allocations could be reduced by reusing the memory that was assigned to drx during the first iteration for the subsequent iterations. Memory can only be reused if the size of drx is greater than or equal to the output of scal_minus. Hence, the functions will have to be modified to perform a check for ensuring memory can be reused. Moreover, the function signature will have to to be modified to add a reference to the array to which the output is assigned, in this case, drx. Another alternative and is to implement a specialised function for the optimisation which satisfies the above mentioned criterion. We chose the second alternative for the optimisation.

```
for(k=1;k<= n;k=k+static_cast<long>(1)) {
  drx = BlasDouble::scal_minus(Rx,vr_temp28);
}
```

Listing 6.6 An example of an array operation which is optimised

6.3.1 Supported Functions

Since a check for sufficient memory allocation needs to be made inside the function, a reference to the output array also needs to the be passed. Hence we implement specialised functions for this optimisation. The Supported library functions include many of the array operations described in Subsection 4.1.4 and a few other library functions. For dimension collapsing functions we support cases where a scalar value is returned. Table 6.3 gives a list of functions for which an implementation support the memory optimisation exists.

Function Name	Function description	Scalar version
mmult	Matrix multiplication	Yes
scal_mult	Scalar Matrix Multiplication	No
vec_add	Array Addition	No
vec_copy	Array Copy	No
vec_sub	Array Subtraction	No
scal_add	Scalar Array Addition	No
scal_minus	Scalar Array Subtraction	No
transpose	Matrix Transpose	No
sum	Sum of Array Elements	Yes
mean	Mean of Array Elements	Yes
sliceArray	Get array slice	No

Table 6.3 List of functions that support memory optimisation

6.3.2 Checking for Sufficient Memory

As mentioned before, the specialised functions accept a reference to the output array as an input parameter. The output array is then checked to determine whether the maximum number of elements that the array can hold is greater than or equal to the number of ele-

Array operation without optimisation	Array operation with optimisation
`drz = BlasDouble::scal_minus(Rz, vr_temp30);`	`BlasDouble::scal_minus(Rz, vr_temp30,&drz);`

Table 6.4 Table shows the generated code with and without memory optimisations

ments of the output of the operation performed by the function. The number of elements are calculated by taking the product of the dimensions of the array. If the memory is sufficient, no memory is allocated to the array whereas memory is allocated if it is not sufficient. In either case, the dimensions are modified to be equal to the expected dimensions of the output of the array operation.

6.3.3 Code Generation

While generating code for assignment statements, the compiler checks for library call expressions which can be compiled to specialised function calls. The compiler does this by checking the function name against a hash set which stores a list of functions that can be specialised. The compiler generates the specialised function call in place of the assignment statement. It then passes the reference LHS of the assignment statement as a parameter to the function.

Table 6.4 gives an example of the generated codes with and without the memory optimisations. The left column shows the function call without the optimisation and the right column shows the function call with the optimisation. The example shows an array operation scal_minus which subtracts a scalar vr_temp30 from every element in the array Rz and assigns the result to drz. The optimisation passes the output array drz's reference to the function where a check for sufficient memory is performed.

Chapter 7
Results

A major goal of the thesis was to improve the performance of array-based languages like MATLAB and Python's NumPy library by compiling computationally intensive functions to C++. To demonstrate these performance results, we compared the performance of the generated code with that provided by various tools for scientific computing. Seventeen MATLAB benchmarks and nine Python benchmarks were used to perform this comparison. Different variations of generated code that can be generated by turning optimisations on and off were also tested.

In this chapter, we give a brief description of the benchmarks that were used to test the performance followed by the results themselves and our analysis of these results.

7.1 Benchmarks

Two separate set of benchmarks were used for MATLAB and Python.

7.1.1 MATLAB Benchmarks

The MATLAB benchmarks used for the performance were obtained from various sources. The sources include the FALCON project[DRP99] , the OTTER project [QMSZ98], Chalmers university of technology[1], Mathworks central file exchange[2] and the presentation on par-

[1] http://www.elmagn.chalmers.se/courses/CEM/
[2] http://www.mathworks.com/matlabcentral/fileexchange

allel programming in MATLAB by Burkhadt and Cliff[3]. The benchmarks cover commonly occurring MATLAB features such as builtin function calls, array indexing including slicing operations and array operations like array addition, matrix multiplication, etc. Table 7.1 gives the list of benchmarks used along with their descriptions and source.

Benchmark	Source	Description
bbai	MATLAB file exchange	Implementation of the Babai estimation algorithm
bubble	McLab	Bubble Sort
capr	Chalmers University	Computes the capacitance of a transmission line using fine difference and Gauss-seidel
clos	Otter project	Calculates the transitive closure of a directed graph
crni	Falcon project	Crank-Nicholson solution to the heat equation
dich	Falcon project	Dirichlet solution to Laplace's equation
fiff	Falcon project	Computes the finite difference solution to the wave equation
ldgr	-	Calculates derivatives of Legendre polynomials
mbrt	McFor project	Computes Mandelbrot sets
nb1d	Otter project	Simulates the 1-dimensional n-body problem
matmul	McLab	naive matrix multiplication
mcpi	McLab	Calculates π by the Monte Carlo method
numprime	Burkardt and Cliff	Simulates the sieve of Eratosthenes for calculating number of prime numbers less than a given number
scra	ACM CALGO	Implementation to produce a reduced-rank approximation to a matrix
spqr	ACM CALGO	Implementation to compute a pivoted semi-QR decomposition of an m-by-n matrix A
quadrature	Burkardt and Cliff	Simulates the quadrature approach for calculating integral of a function

Table 7.1 List of MATLAB Benchmarks used for experiments

7.1.2 Python Benchmarks

Many of the Python benchmarks are Python ports of the Ostrich benchmark suite[KFBK+14]. The benchmarks contain scalar operations as well as array index operation. Six of the nine Python benchmarks support parallelism. Table 7.2 gives the list of Python benchmarks that were used. The Python ports of the Ostrich benchmark suite are known as PyDwarfs. The rest of the benchmarks are part of a suite that were put together by the open-source Python

[3] `http://people.sc.fsu.edu/~jburkardt/presentations/matlab_parallel.pdf`

community focused on compilers. The suite is known as NumFocus and can be found on github[4].

Benchmark Name	Source	Description
arc_distance	NumFocus	Calculates the pairwise arc distance between all points in vector a and b.
fft	Pydwarfs	Fast Fourier Transform
growcut	NumFocus	Implementation of GrowCut segmentation
julia	NumFocus	Calculates the Julia fractal
lud	PyDwarfs	LU decomposition factors a matrix as the product of a lower triangular matrix and an upper triangular matrix
pagerank	PyDwarfs	PageRank is a link analysis algorithm used by Google Search
pairwise	NumFocus	Computes the pairwise distance between a set of points in 3D space.
spmv	PyDwarfs	Sparse Matrix-Vector Multiplication
srad	PyDwarfs	Tracks the movement of a mouse heart over a sequence of 104 609x590 ultrasound images to record response to the stimulus

Table 7.2 List of Python Benchmarks used for experiments

7.2 Experimental Setup

We ran separate experiments for MATLAB and Python benchmarks. Different variations of the code generated by Velo**C**ty were also tested. Table 7.3 gives a list of variations generated by Velo**C**ty. All the benchmarks were tested on a machine running GNU/Linux(3.8.0-35-generic #52-Ubuntu) with a Intel(R) Core(TM) i7-3820 CPU @ 3.60GHz with 16GB of memory. We also ran experiments on different compiler tools developed for both MATLAB and Python. Each version of the benchmarks was executed 10 times and the average execution time was recorded. The following subsections describe the tools for each language against which the different versions of Velo**C**ty were compared and explain the aspects of the experimental setup that are specific to each language.

[4] `https://github.com/numfocus/python-benchmarks`

Variation name	Description
Baseline VeloCty	Generated C++ code without optimisations and with array bounds checks enabled
VeloCty no-checks	Generated C++ code without optimisations and without array bounds checks.
VeloCty memory optimisation	Generated C++ code with memory optimisations and with array bounds checks enabled.
VeloCty bounds check optimisation	Generated C++ code with boundscheck optimisations and with array bounds checks enabled.
VeloCty parallel	Generated C++ code with parallel constructs and with array bounds checks enabled.
VeloCty all optimisations	Generated C++ code with all optimisations and with array bounds checks enabled.

Table 7.3 The table gives a list of benchmark variations generated by Velo**C**ty along with the description of each

7.2.1 Experimental Setup for MATLAB

In order to gauge Velo**C**ty's performance against current compiler tools for MATLAB, the MATLAB benchmarks were executed on the Mathworks' 2014b release of the MATLAB interpreter and JIT compiler. We also used the Mathworks' MATLAB-Coder implementation to compile the benchmarks to C++. This generated C++ code is compiled as a dynamic library similar to the method used by Velo**C**ty. Both Velo**C**ty and MATLAB-coder use the MEX compiler to compile the C++ code and generate the shared library. MEX internally uses the g++-4.6.4 compiler.

7.2.2 Experimental Setup for Python

Similar to MATLAB, we gauged our performance of the Velo**C**ty code against existing compiler tools for Python. We used the reference C-Python interpreter version 3.2.3 and Cython[Cyt] version 0.21, which is a compiler used to generate C-extensions for Python. Both, Velo**C**ty and Cython use g++-4.6.4 through distutils for compilation.

7.3 MATLAB Results

7.3.1 Overall Results

We ran experiments on 17 MATLAB benchmarks. We compared the VeloCty with all optimisations enabled with the Mathworks' MATLAB implementation and MATLAB-coder. We measured the speedup of the VeloCty backend and the MATLAB-coder versions compared to Mathwork's MATLAB JIT compiler. *Figure 7.1* shows a bar graph with the results of the experiment. The red bars show the speedup of MATLAB-coder and the blue bars show the speedup of the VeloCty backend with all optimisations enabled. The geometric mean for the speedup of the VeloCty version was 8.05x as compared to the geometric mean of 3.89x for the MATLAB-coder version. The largest speedup was shown by the quadrature benchmark. The benchmark was 458x times faster than Mathworks' MATLAB. The benchmark consists of operations on scalar operations and hence gives a high speedup. The smallest speedup of 1.31x, is given by the closure benchmark. The benchmark's computationally intensive code section is a while loop containing a matrix multiplication operation. All three versions, the VeloCty backend, Mathwork's MATLAB and MATLAB-coder use the Intel MKL BLAS library and hence show similar performance.

For most benchmarks, our VeloCty backend was faster than MATLAB-coder. The benchmarks, bbai, lgdr, nb1d, fft and numprime are exceptions. Lgdr bbai and nb1d contain array slicing and array operations which do not internally make calls to the BLAS library and hence take longer to execute compared to MATLAB-coder. The fft benchmark contains a loop whose direction can not be identified at run time and hence a loop vector needs to be initialised and iterated over as described in Subsection 4.5.2. numprime contains scalar operations and a square root function call. The square root function's MATLAB implementation may be faster than the standard C++ implementation and hence the numprime benchmark performs in MATLAB-coder than in the VeloCty version.

As we can also observe, MATLAB-coder shows a positive speedup on all benchmarks except for mcpi. The reason for this is that the benchmark contains a loop with calls to the random function inside the body. These functions return a single scalar value. However, in case of MATLAB-coder, an 1x1 matrix is returned. Since a heap allocation is needed for

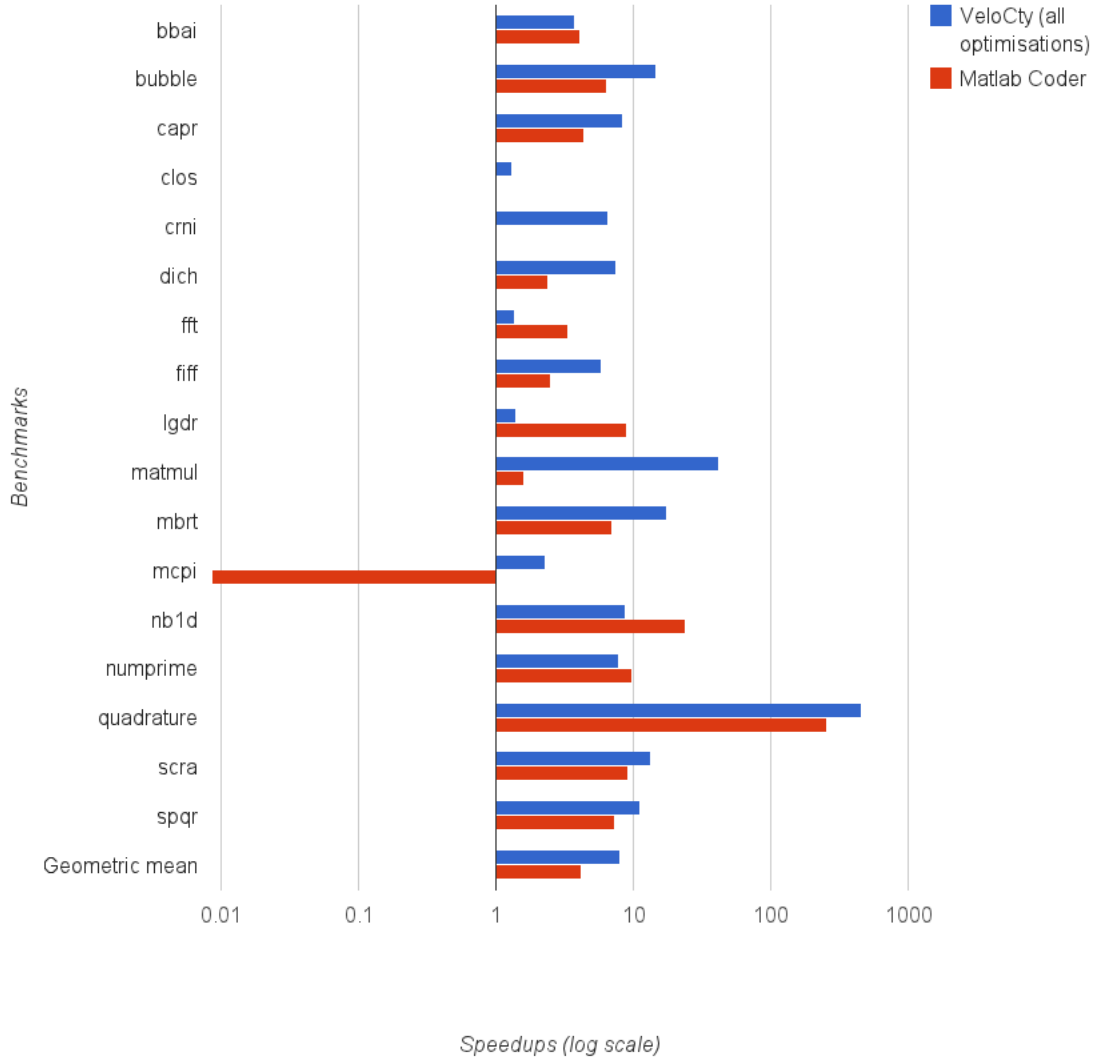

Figure 7.1 The bar graph gives the speedups of the Velo**C**ty backend with all optimisations enabled and MATLAB-coder compared to the Mathworks' JIT and VM implementation. Higher is better

every iteration, the benchmark is significantly slower than Mathworks' MATLAB.

The crni benchmark is an example of a feature that is supported by Velo**C**ty but not by MATLAB-coder. The benchmark contains a growing array which is not supported by MATLAB-coder. Hence, a MATLAB-coder version of the benchmark cannot be generated.

7.3.2 Impact of Array Bounds Checks on Performance

Table 7.4 gives the slowdown of the generated VeloCty code because of bounds checks. This experiment allowed us to determine the effect bounds check had on performance. The table lists the slowdown of the baseline Velo**C**ty code compared to the Velo**C**ty code with checks disabled. 6 benchmarks show a slowdown of 1.5x or higher. These benchmarks are bubble, capr, dich, fft, fiff and matmul. The geometric mean of the slowdown for all benchmarks is 1.66x. If only the geometric mean of the 6 benchmarks that slowed down significantly are considered, we get a geometric mean of 3.66x. The matmul benchmark shows the highest slowdown with 10.44x. The slowdown for the crni benchmark cannot be calculated since a version of the generated code without checks cannot be executed.

7.3.3 Impact of Bounds Check Optimisations on Performance

The previous experiment showed us that bounds checks have a significant impact on performance. This was the reason we implemented an optimisation to eliminate the bounds checks where possible. In order to determine the improvement in performance when bounds check optimisations were enabled, we compared the speedup of the generated VeloCty code with bounds check optimisation enabled against the baseline Velo**C**ty version. Table 7.5 gives the speedups for all the benchmarks. The geometric mean of speedups with the optimisation enabled is 1.62x. If the speedups of the benchmarks that show a significant speedup are considered, we see a speedup of 3.18x. The fft benchmark does not show an improvement in performance. This is because the benchmark contains a loop whose direction can not be determined and hence the bounds checks inside the loop body can not be moved outside.

Benchmarks	Slowdown
bbai	1.21
bubble	8.61
capr	2.62
clos	0.84
crni	-
dich	1.71
fft	1.43
fiff	4.15
lgdr	1.01
matmul	10.44
mbrt	1.02
mcpi	1.00
nb1d	1.05
numprime	1.06
quadrature	1.00
scra	1.99
spqr	1.04
Geometric mean	1.66
Geometric mean (Affected Benchmarks)	3.66

Table 7.4 The table lists the slowdowns of the VeloCty baseline code compared to when Velo**C**ty-no-checks

7.3.4 Impact of Memory Optimisations on Performance

As we had already identified that many of the benchmarks contain array operations inside a loop where memory is allocated continuously. As dynamic memory allocations are expensive, we implemented an optimisation to eliminate unnecessary memory allocations and instead reuse previously allocated memory where possible. The performance improvement of the generated code when the memory optimisations were enabled was also gauged. We calculated the speedups of the generated VeloCty code with memory optimisations enabled compared to the baseline VeloCty code for all the benchmarks. Table 7.6 gives the speedups for the benchmarks. The geometric mean of the speedups for all benchmarks is 1.14x. Four benchmarks, capr, nb1d, scra, spqr showed speedups of 1.54x, 1.59x, 3.57x and 2.77x respectively. All of these benchmarks consisted of loops inside which array operations were

Benchmarks	Speedups
bbai	1.07
bubble	7.84
capr	2.49
clos	1.00
crni	2.23
dich	1.71
fft	1.01
fiff	4.12
lgdr	1.02
matmul	10.54
mbrt	1.00
mcpi	1.00
nb1d	1.06
numprime	0.99
quadrature	1.00
scra	1.04
spqr	1.01
Geometric mean	1.62

Table 7.5 The table lists the speedups obtained when the array bounds check optimisations are turned on against the baseline Velo**C**ty code

performed. Since, because of the optimisation, previously allocated memory was reused, we observed a noticeable speedup. The geometric mean of the four affected benchmarks is 2.36x.

7.3.5 Impact of Parallel Execution of VeloCty Code

Three of the MATLAB benchmarks, nb1d, matmul and mbrt can be executed in parallel. We calculated the speedups of the three benchmarks in parallel compared to the baseline VeloCty version and the speedups of the benchmarks executed using the Mathworks' Parallel Computing Toolbox[Matd] compared to the Mathwork's MATLAB JIT executing code sequentially.

Table 7.7 gives the speedups for the three benchmarks. In the case of the Velo**C**ty parallel version, matmul and mbrt benchmarks show significant speedups of 3.87x and

Benchmarks	VeloCty Memory optimisation
bbai	1.18
bubble	1.00
capr	1.54
clos	1.00
crni	1.10
dich	1.00
fft	1.00
fiff	1.14
lgdr	1.08
matmul	0.99
mbrt	1.00
mcpi	1.00
nb1d	1.59
numprime	1.00
quadrature	1.00
scra	3.33
spqr	2.77
Geometric mean	1.23
Geometric mean(Affected Benchmarks)	2.36

Table 7.6 The table lists the speedups for the different MATLAB benchmarks when memory optimisations are enabled compared to baseline Velo**C**ty code

3.26x respectively. This is because in the case of the the two benchmarks, a very small portion of the code needs to be executed sequentially. Moreover, parallel portion of the code is computationally intensive thus making the thread management time a small portion of the total execution time. On the other hand, the nb1d benchmark shows no speedup. The parallel version is 1.01 times faster than the baseline Velo**C**ty. This can be attributed to the fact that the loop being parallelised is nested inside another loop executing sequentially. Moreover, the loop being executed in parallel is not computationally intensive and hence the benchmark does not benefit from parallel execution.

On the other hand, the Parallel Computing Toolbox shows a speedup of 3.35x compared to the sequential MATLAB version. This algorithm is embarrassingly parallel and has little data transfer which is ideal for the toolbox's multiprocessing based model. The matmul benchmark shows a speedup of 1.05x. The smaller speedup may be attributed to higher

number of data transfers. The benchmark nb1d shows a slowdown when executed in parallel which may be because of the fact that since only the inner loop is executed in parallel, the more time is spent in the master process in data management than in code execution.

Benchmarks	Speedup Velocty parallel v/s VeloCty Baseline	Speedup MATLAB Parallel v/s MATLAB Sequential
matmul	3.87	1.05
mbrt	3.26	3.35
nb1d	1.01	0.32

Table 7.7 The table lists the speedups of the benchmarks in VeloCty code with parallel execution over baseline VeloCty code.

7.3.6 Summary of MATLAB Results

Figure 7.4 shows a chart with the speedups of the different versions of Velo**C**ty code for the MATLAB benchmarks compared to the Mathworks' MATLAB interpreter and VM. The blue bars represent the speedups of baseline Velo**C**ty, the red bars indicate the speedups of the Velo**C**ty versions with bounds check optimisations turned on, the yellow bars indicate the speedups of the Velo**C**ty versions with both memory and bounds check optimisations turned on and the green bars indicate the speedups of the Velo**C**ty versions with bounds check optimisation, memory optimisation and parallel code execution.

As we can observe, we see an increase in the geometric means as we add optimisations. The geometric mean for baseline Velo**C**ty is 3.76x, the geometric mean for the Velo**C**ty version with bounds checks optimisations enabled was 6.10x, that for Velo**C**ty with bounds checks and memory optimisations was 8.24x and finally the version with all optimisations was 8.5x. In case of the Velo**C**ty versions with all the optimisations, if we only consider the three benchmarks that are executed in parallel, we observe a geometric mean of 18.27x.

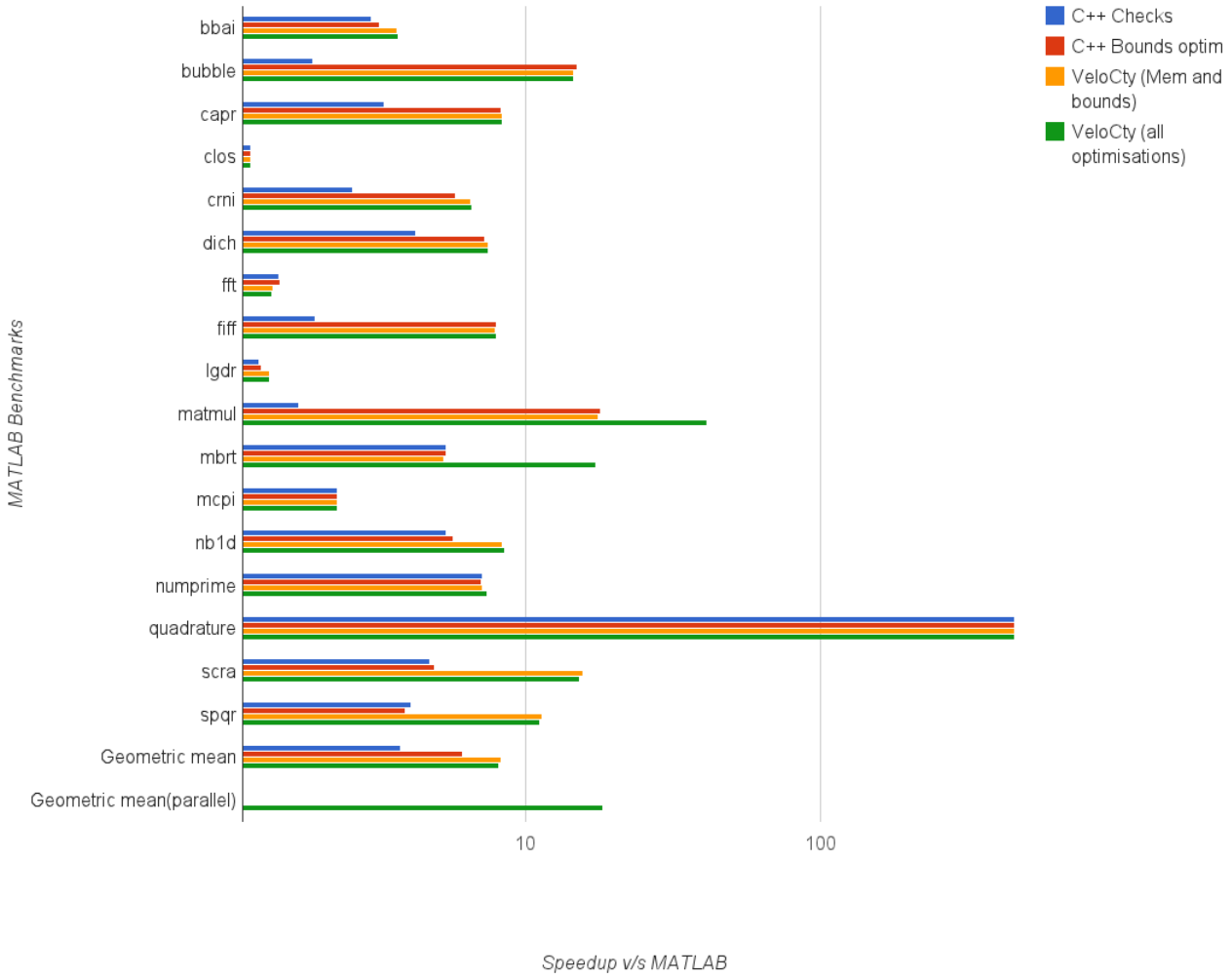

Figure 7.2 The figure compares the speedups of different Velo**C**ty versions against the Mathworks' interpreter and JIT compiler version for the MATLAB benchmarks. Geometric mean(parallel) gives the geometric mean of three benchmarks matmul, mbrt and nb1d when they are executed with all optimisations.

7.4 Python Results

7.4.1 Overall Results

We ran experiments on 9 Python benchmarks. Similar to the experiments on MATLAB benchmarks, we compared the generated VeloCty code without bounds checks to Cython and the CPython interpreter. *Figure 7.3* is a bar graph showing the speedup of the generated VeloCty code with checks disabled, the generated VeloCty code with checks enabled and the Cython code compared to the CPython interpreter. The blue bars indicate the speedup of the generated VeloCty code with checks enabled and the red bars indicate the speedup of the Cython code. The geometric mean of the speedups for the generated VeloCty code without checks was 397.17x. The largest speedup of 1281.67x was shown by the lud benchmark. The smallest speedup of 40.98 was shown by the fft benchmark. Note that the Python results are compared against a pure interpreter, C-Python, whereas in case of the MATLAB, the results were compared against a interpreter with a JIT compiler. A JIT compiler gives better performance compared to a pure interpreter. Hence we see higher speedups for the VeloCty code for Python as compared to the ones we observed for MATLAB.

The baseline VeloCty code is faster than the Cython code for all the benchmarks. Comparing the speedup of our generated VeloCty code to the Cython code, a mean speedup of 2.21x was found. The largest speedup of 14.93x was shown by the fft benchmark. The benchmarks arc_distance, lud and pagerank take the same time to execute as our generated C++ code. All three benchmarks have fewer array index operations and more scalar operations compared to the other benchmarks. On the other hand the fft benchmark performs significantly better for the generated VeloCty code than the Cython version. The fft benchmark contains recursive function calls. Cython adds checks to ensure validity of the input arguments of a function as well as validity of the arguments being passed from the function call point.

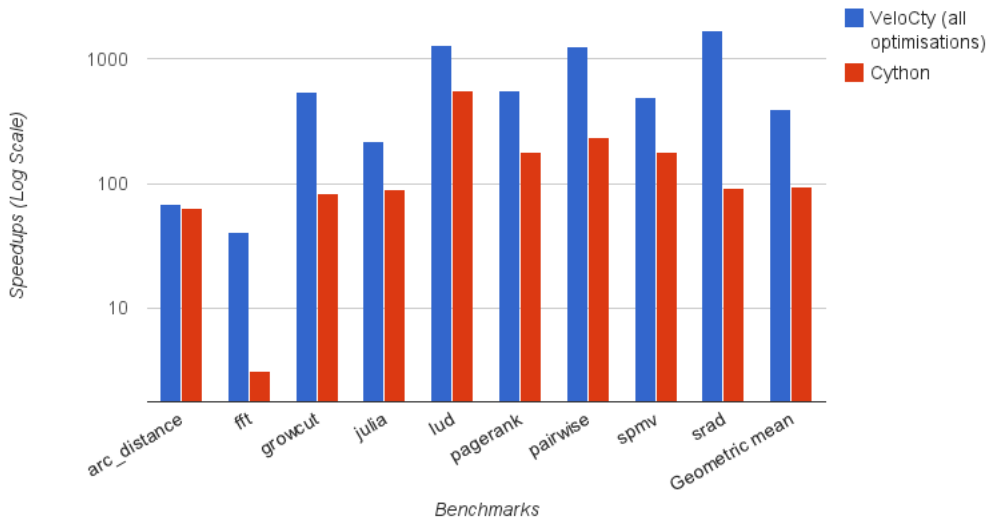

Figure 7.3 The figure compares the speedups of the Velo**C**ty code with all optimisations enabled and and speedups of Cython against the C-Python version for the Python benchmarks.

7.4.2 Impact of Array Bounds Checks on Performance

Enabling array bounds checks gives a significant slowdown in 4 of the 9 benchmarks. These benchmarks are growcut, pairwise, spmv and srad. Table 7.8 lists the slowdowns observed for the generated VeloCty versions with checks enabled compared to the generated VeloCty version with checks disabled. The geometric mean of the slowdown is 1.27x. The geometric mean of the slowdown for the affected benchmarks was 1.63. The highest slowdown was shown by the pairwise benchmark. Benchmarks which were not affected by array bounds checks enabled were ones which contained loops with fewer loop iterations or with fewer array index operations in their bodies.

Benchmarks	Slowdown
arc_distance	1.02
fft	1.09
growcut	1.28
julia	1.01
lud	1.03
pagerank	1.01
pairwise	1.94
spmv	1.73
srad	1.64
Geometric mean	1.26
Geometric mean(Affected Benchmarks)	1.63

Table 7.8 Slowdown of the Python benchmarks for VeloCty code with checks enabled compared to VeloCty code without checks

7.4.3 Impact of Bounds Check optimisations on benchmark performance

We also timed versions of the generated VeloCty code with the bounds check optimisations enabled. We calculated the speedup of Velo**C**ty with bounds check optimisations against the baseline Velo**C**ty. Table 7.9 lists slowdowns for all the Python benchmarks. The geometric mean of slowdown for the VeloCty code with optimisation is 1.22x. The geometric mean of the speedups of benchmarks which showed a significant speedup is 1.79x. Maximum speedup was of 2.00x and the smallest was of 1.59x.

7.4.4 Impact of parallel execution of VeloCty code

6 of the 9 benchmarks could be executed in parallel. We calculated the speedups of the VeloCty versions executing in parallel with the baseline VeloCty version. The geometric mean of the speedups was 2.50x. Maximum speedup was observed in the growcut benchmark. The benchmark showed a speedup of 3.96x. The smallest speedup of 2.20x was observed by the srad benchmark. Table 7.10 gives the speedups for the 6 benchmarks that could be executed in parallel.

Benchmark	VeloCty Bounds Check Optimisation
arc_distance	1.02
fft	1.04
growcut	1.06
julia	1.01
lud	1.02
pagerank	1.02
pairwise	2.00
spmv	1.59
srad	1.60
Geometric mean	1.22

Table 7.9 Speedup of VeloCty with check optimisation and baseline VeloCty.

Benchmark	VeloCty Parallel
arc_distance	1.02
fft	1.04
growcut	3.96
julia	1.01
lud	2.41
pagerank	2.73
pairwise	2.34
spmv	1.83
srad	2.20
Geometric mean	1.86

Table 7.10 The table lists the speedups of the VeloCty parallel versions of the Python benchmarks against baseline Velo**C**ty code

7.4.5 Summary of Python results

Figure 7.4 shows a chart with the speedups of the different versions of Velo**C**ty code for the Python benchmarks compared to the C-Python interpreter. The blue bars represent the speedups of baseline Velo**C**ty, the red bars indicate the speedups of the Velo**C**ty versions with bounds check optimisations turned on and the yellow bars indicate the speedups of the Velo**C**ty versions with bounds check optimisation and parallel code execution.

Similar to MATLAB we can see the geometric mean of the Velo**C**ty versions increasing

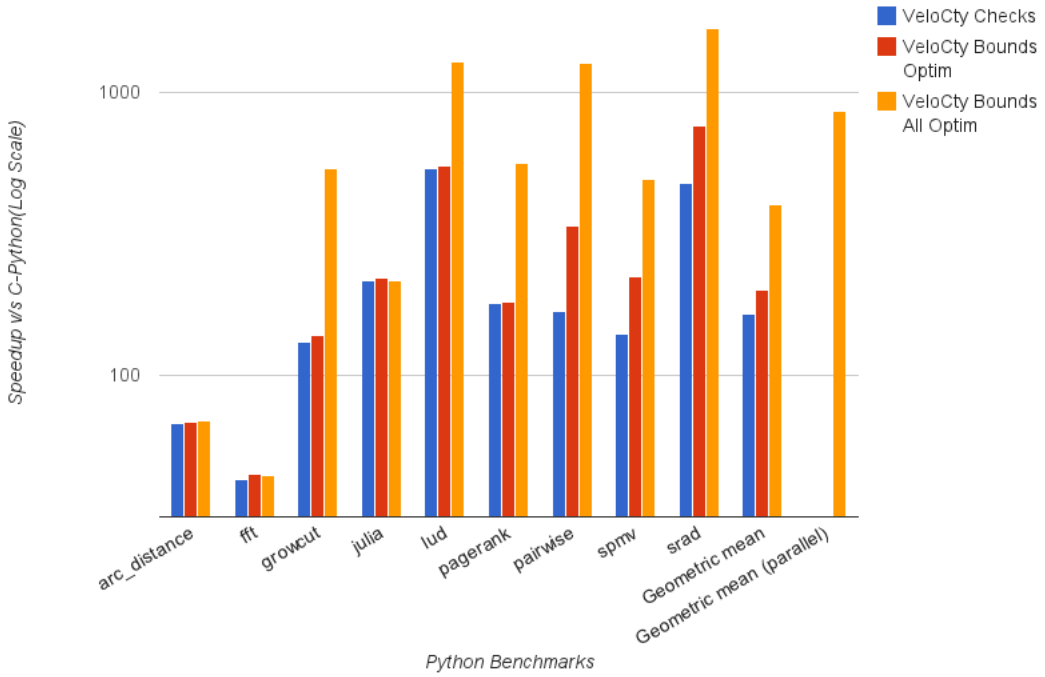

Figure 7.4 The bar graph gives the speedups of the Velo**C**ty backend for different Velo**C**ty versions compared to the Mathworks' JIT and VM implementation. Higher is better. Geometric mean(parallel) gives the speedups of the 6 benchmarks that can be executed in parallel.

as we add optimisations. The geometric mean of the baseline Velo**C**ty version is 164.48x , the geometric mean of the version with bounds check optimisations enabled is 200.8x and the geometric mean when Velo**C**ty code is executed in parallel is 400.98x. The geometric mean of benchmarks which are executed in parallel is 860.09x.

Note that since none of the Python benchmarks contained any array slicing operations or array operations, performance would not differ when the memory optimisation was used and hence we do not specify those numbers in this section.

7.5 Summary

Through our experiments we showed that compiling 'hot' functions of MATLAB and Python show significant performance improvement. For the MATLAB benchmarks, we observed significant speedups compared to the Mathworks' MATLAB interpreter and JIT compiler. We also observed comparable performance with respect to MATLAB-coder. We identified bottlenecks in generated code and implemented optimisations, namely the bounds check elimination and the memory optimisation to reduce the impact of the bottlenecks. Due to these optimisations we also observed significant speedups over MATLAB-coder for most benchmarks. Moreover we also identified the bottlenecks in the benchmarks which showed poorer performance compared to MATLAB-coder and suggested optimisations to eliminate the bottlenecks. The generated code showed significant performance gains for three benchmarks when it was executed in parallel using OpenMP.

On the Python side, we saw very large speedups against the C-Python interpreter. The larger speedup could be attribute to the fact that comparison was against an interpreter without a JIT compiler. We also observed equal or better performance when compared to Cython. The performance of the generated code improved further when the optimisations were added and the code was executed in parallel.

In conclusion, the VeloCty code with all optimisations enabled generated by VeloCty from MATLAB is 8.05 times faster than the Mathwork's MATLAB. The generated VeloCty code for Python is 400.98 times faster than the C-Python interpreter. We believe that these results are encouraging and have motivated us to further develop the compiler for higher performance gains.

Chapter 8
Related Work

Over the years, many dynamic languages were developed which were either developed for scientific computing or provided libraries to implement the same. This list includes but is not restricted to MATLAB [Matb] and NumPy[Dev]. Additionally, improving the performance of dynamic languages through ahead of time compilation or just in time compilation has been the interest of researchers for many years. Hence many projects, industry-based, academic and community-based, have been implemented. In this chapter, we discuss some languages which can be viewed as alternatives to MATLAB and NumPy, followed by different tools, similar to VeloCty, which aim to improve the performance MATLAB and NumPy.

8.1 Alternatives to MATLAB and NumPy

A few examples of open-source alternatives to MATLAB and NumPy are Julia[BKSE12], Scilab[INR], R[Fouc] and Octave[GNUb]. Julia is a high-performance dynamic language for high-performance computing. Julia supports distributed-parallel execution and a high-performance library for numerical computing. Scilab is an open source software for numerical computing. Scilab supports high-level 2D and 3D visualisation functions etc. R is a language for statistical computing. R is an alternate implementation of the S language. Octave is an open-source implementation of MATLAB. It supports most of the MATLAB code. Although it was previously interpreted, a JIT compiler was added in version 3.8.1.

8.2 Tools for NumPy

8.2.1 Cython

Cython is a programming language for writing C extensions for Python. Cython was originally based on the Pyrex project[Erw]. Cython allows optional static declarations. This allows C semantics, which are static and fast to be applied for parts of the code instead of dynamic Python semantics. Cython generated code can also make calls into C libraries. Listing 8.1 gives an example of the Cython code for the benchmark arc_distance. Types are provided using the cdef prefix.

```
1
2  def pairwise_kernel(np.ndarray[double,ndim=2] data):
3  cdef int n_samples = data.shape[0]
4  cdef int n_features = data.shape[1]
5  cdef double tmp, d
6  cdef np.ndarray[double,ndim=2] distances = np.empty((n_samples, n_samples))
7  for i in range(n_samples):
8    for j in range(n_samples):
9      d = 0.0
10     for k in range(n_features):
11       tmp = data[i, k] - data[j, k]
12       d += tmp * tmp
13     distances[i, j] = sqrt(d)
14 return distances
```

Listing 8.1 The Cython code with static type annotations that is taken as input by Cython to generate C code. The example is of the arc_distance benchmark

Unlike Velo**C**ty, Cython can only generate C code from Python. Moreover, Cython inserts additional checks for types and memory management. Velo**C**ty on the other hand assumes that all type annotations provided are correct and hence does not place checks.

8.2.2 Numba

Numba[Ana] is a library for Python that can perform JIT compilation given a few annotations. Numba generates machine code using the LLVM[LA04] infrastructure. Numba

also has a CUDA[NBGS08] backend which is currently experimental. Numba can also perform static compilation using the pycc tool that is provided with the library. Numba is a comparatively new project with the initial release in 2012.

8.2.3 Theano

Theano[BLP+12] is a Python library that allows users to define mathematical expressions and then optimises and generates C code dynamically. Theano combines various aspects of computer algebra systems with an optimising compiler. It also generates CUDA code for GPUs. Theano has tight integration with NumPy.

It would be interesting to compare VeloCty's performance with both Theano and Numba in the future. Note that we do compare our performance against Cython.

8.3 MATLAB Tools

8.3.1 MATLAB-coder

The Mathworks' MATLAB-coder[Mata] is a tool to compile MATLAB functions to C/C++. MATLAB-coder accepts the MATLAB function as well as types and shapes for the input arguments of the array and generates C/C++ code. MATLAB-coder offers 3 different options for compilation. The user can generate a standalone C/C++ executable, a C/C++ shared library or a MEX[Matc] function that can be called from MATLAB. Similar to Cython, MATLAB-coder adds type and memory checks which are added by VeloCty.

8.3.2 Falcon

The Falcon project[DRP99] is a MATLAB to Fortran90 compiler. Falcon implements type inference algorithms that were developed for the APL[Bud83, JB00, WS81] language and the SETL[Sch75] language. The compiler inlines all functions and scripts into a single function. Falcon uses a static single assignment(SSA) based intermediate representation. Additionally, they also collected a set of benchmarks for their experiments. We use many of these benchmarks for our experiments.

8.3.3 MaJIC

MaJIC[AP02] stands for MATLAB Just in Time Compiler. It is a continuation of the Falcon project. It performs three different types of optimisations: Source level optimisations for matrix operations, JIT compilation, so that the effects of the 'wild' MATLAB features are reduced and lastly specialised optimisations for sparse matrices.

8.3.4 MENHIR

MENHIR[CB99] is a retargetable compiler for MATLAB that can compile either C or Fortran given a target system description(MTSD). An efficient code is generated that exploits optimised sequential and parallel libraries. The MTSD is used to generate the optimised code for a specific platform.

8.3.5 Mc2For

Mc2For[LH14] is a source to source compiler which transforms MATLAB code to equivalent Fortran code. Mc2For was also developed using the *Mc***LAB** framework. Even though Mc2For is a ahead of time compiler similar to Velo**C**ty, Mc2For compiles complete MATLAB programs instead of 'hot' code sections. Hence if any part of the MATLAB program has constructs that cannot be compiled ahead of time, the entire program cannot be compiled using Mc2For.

8.3.6 MiX10

MiX10[Kum14] is a source to source compiler which compiles MATLAB code to X10[CGS$^+$05]. X10 is a high-performance language developed at IBM. MiX10 compiles dynamically typed MATLAB code to a statically typed X10 language. The X10 compiler itself compiles the X10 code to C++ and Java. Similar to Mc2For compiles complete MATLAB programs to X10 and hence cannot compile MATLAB programs having dynamic constructs.

Chapter 9
Conclusions and Future Work

9.1 Conclusions

The aim of the thesis was to improve performance of array based-languages by compiling computationally intensive code-sections to C++ and then compiling them to a shared library that can be called from the source array-based language. A partial compilation ensures that users can continue writing code in the source language. Another advantage of partial compilation is that portions of code which cannot be compiled ahead of time can be skipped.

We used the Velociraptor toolkit for implementing the compiler. Velociraptor provides language agnostic tools and analyses to aid generation of high-performance code. The Velociraptor intermediate representation, VRIR, is a high-level AST based representation which has semantics that are close to those of array-based languages. This makes writing a front-end compiler to VRIR easy. Moreover, VRIR has flexible semantics to accommodate the semantic differences of different array-based languages. This allows us to write a single backend to compile from VRIR to C++, which can be used for multiple array-based languages. VRIR also contains constructs such as parallel for, map and reduce statements which can be used to generate parallel code. Additionally, Velociraptor provides analyses and transformations which aid in code generation.

Contributions of this thesis can be divided into four parts. The first is the implementation of a frontend for the MATLAB language. The frontend was implemented using the open source *McLAB* toolkit. We faced challenges while compiling from a MATLAB-specific intermediate representation to a language agnostic one. Determining the types of the expression nodes in VRIR and converting a colon expression to a range expression include some of them. The second is the generation of the glue code that is required to interface the generated code with MATLAB. This involved the generation of code that converts MATLAB-specific arrays to Velo**C**ty arrays, calling the generated function and finally converting the Velo**C**ty arrays back to the MATLAB-specific array. The implementation of a compiler backend from VRIR to C++ is the third contribution of this thesis. The code generator was flexible enough to generate C++ code for the different semantics supported by VRIR. For example, the code generator can generate code for both row major and column major arrays. Additionally, we implemented runtime libraries for both MATLAB and Python. These libraries implement different builtin functions supported by both languages, array bounds checks and array slicing operations among other functions. The final contribution of the thesis was to optimise the generated code. We implemented optimization to eliminate bounds checks inside loop bodies and eliminated redundant unnecessary memory allocations during array operations. Also, we supported naive parallelism using OpenMP.

We observed significant gains in performance when comparing the generated code against the standard implementations for MATLAB and Python, the Mathworks' MATLAB interpreter and JIT compiler and the CPython interpreter respectively. We also observed gains over other tools for performance improvement for these languages such as the MATLAB-coder for MATLAB and Cython for NumPy.

In conclusion, we would like to state that Velo**C**ty does achieve significant performance gains for both MATLAB and NumPy. We believe that partial compilation improves usability by allowing users to continue using their preferred scientific language. Finally, our compiler backend is language agnostic and can help compiler writers improve performance of other languages such as R and Julia. Velo**C**ty is open source and freely available and hence can be reused and modified by researchers for their own work.

9.2 Future Work

Although Velo**C**ty does achieve significant performance gains, there still is scope for improvement. We see five areas where improvement in performance may be achieved.

9.2.1 Automatic detection of computationally intensive code sections

In the current implementation of Velo**C**ty, we depend on the user to identify and annotate code sections, which can then be compiled to C++. In the future, the implmentation could be improved to automatically identify computationally intensive code sections and compile them to C++.

9.2.2 GPU code generation

Heterogeneous architectures are gaining popularity in recent times. Many low-level languages and libraries have been developed for writing code for these architectures. These languages make good targets for high-performance compilers. An enhancement of Velo**C**ty could be the generation of GPU code from the parallel for loops.

9.2.3 Auto-parallelization

Velo**C**ty currently supports naive parallelism. The user has to annotate the parallel for loops with a list of variables that are shared inside the loop. A possible improvement to Velo**C**ty would be the implementation of algorithms that automatically identify loops which can be executed in parallel and identify the list of variables that are shared and ones that are private. Another improvement would be identification of statements that can be vectorised and replacing the statements by vector instructions.

9.2.4 Optimisations

Many optimisations can be performed on the generated code to improve its performance. The MATLAB frontend can be optimised to eliminate copy statements and copies of arrays during function calls when the arrays are being written to. The bounds check optimisation can be improved to support a larger range of loops and indices. Additionally, operations on arrays can be made lazy and can only be performed when the array elements are accessed. This optimisation may lead to performance gains, since if a chain of array operations are performed, memory can be reused across operations.

9.2.5 Faster Builtins

The builtin functions in the runtime libraries have naive implementations. Many techniques such as parallelism and vector instructions can be used to improve the performance of these functions.

9.2.6 Readability

The aim of the thesis was to ensure correct compilation of optimised code for MATLAB and NumPy. In the future, we would like to add information to the generated code which would improve readability and simplify debugging of code. One approach could be to add the line numbers of the original code from which a certain code section has been generated.

Bibliography

[Ana] Continuum Analytics. Numba. `http://numba.pydata.org/`.

[AP02] George Almási and David Padua. Majic: Compiling matlab for speed and responsiveness. *SIGPLAN Not.*, 37(5):294–303, May 2002.

[BKSE12] Jeff Bezanson, Stefan Karpinski, Viral B. Shah, and Alan Edelman. Julia: A fast dynamic language for technical computing. *CoRR*, abs/1209.5145, 2012.

[BLP+12] Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, James Bergstra, Ian J. Goodfellow, Arnaud Bergeron, Nicolas Bouchard, and Yoshua Bengio. Theano: new features and speed improvements. Deep Learning and Unsupervised Feature Learning NIPS 2012 Workshop, 2012.

[Bud83] Timothy A. Budd. An apl compiler for the unix timesharing system. *SIGAPL APL Quote Quad*, 13(3):205–209, March 1983.

[CB99] Stéphane Chauveau and François Bodin. Menhir: An environment for high performance matlab. *Sci. Program.*, 7(3-4):303–312, August 1999.

[CGS+05] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: An object-oriented approach to non-uniform cluster computing. *SIGPLAN Not.*, 40(10):519–538, October 2005.

[CLD+10] Andrew Casey, Jun Li, Jesse Doherty, Maxime Chevalier-Boisvert, Toheed Aslam, Anton Dubrau, Nurudeen Lameed, Amina Aslam, Rahul Garg, Soroush Radpour, Olivier Savary Belanger, Laurie J. Hendren, and Clark Verbrugge. Mclab: an extensible compiler toolkit for matlab and related languages. In *C3S2E'10*, 2010, pages 114–117.

[Com] The Scipy Community. Numpy C-API. `http://docs.scipy.org/doc/numpy/reference/c-api.html`.

[Cor] Intel Corporation. Math Kernel Library. `https://software.intel.com/en-us/intel-mkl`.

[Cyt] Cython. cython. `http://cython.org/`.

[Dev] NumPy Developers. NumPy. `http://www.numpy.org/`.

[DH12] Anton Willy Dubrau and Laurie Jane Hendren. Taming matlab. *SIGPLAN Not.*, 47(10):503–522, October 2012.

[DHR11] Jesse Doherty, Laurie Hendren, and Soroush Radpour. Kind analysis for matlab. *SIGPLAN Not.*, 46(10):99–118, October 2011.

[Doh11] Jesse Doherty. Mcsaf: An extensible static analysis framework for the matlab language. Master's thesis, August 2011.

[DRP99] Luiz De Rose and David Padua. Techniques for the translation of matlab programs into fortran 90. *ACM Trans. Program. Lang. Syst.*, 21(2):286–323, March 1999.

[Erw] G. Erwing. Pyrex. `http://www.cosc.canterbury.ac.nz/greg.ewing/python/Pyrex/`.

[Foua] Python Software Foundation. Python. `https://www.python.org/`.

[Foub] Python Software Foundation. Python C-API. `https://docs.python.org/3.2/extending/extending.html`.

[Fouc] R Foundation. The R language. http://www.r-project.org/.

[Foud] Standard C++ Foundation. C++ Language. https://isocpp.org/.

[GH14] Rahul Garg and Laurie Hendren. Velociraptor: An embedded compiler toolkit for numerical programs targeting cpus and gpus. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, Edmonton, AB, Canada, 2014, PACT '14, pages 317–330. ACM, New York, NY, USA.

[GNUa] GNU. FORTRAN Language. https://gcc.gnu.org/fortran/.

[GNUb] GNU. GNU Octave. https://www.gnu.org/software/octave/about.html.

[INR] INRIA. Scilab. http://www.scilab.org/scilab/about.

[JB00] Pramod G. Joisha and Prithviraj Banerjee. Correctly detecting intrinsic type errors in typeless languages such as matlab. *SIGAPL APL Quote Quad*, 31(2):7–21, December 2000.

[KFBK+14] Faiz Khan, Vincent Foley-Bourgon, Sujay Kathrotia, Erick Lavoie, and Laurie Hendren. Using javascript and webcl for numerical computations: A comparative study of native and web technologies. In *Proceedings of the 10th ACM Symposium on Dynamic Languages*, Portland, Oregon, USA, 2014, DLS '14, pages 91–102. ACM, New York, NY, USA.

[Kum] Vineet Kumar. IsComplex analysis in Tamer. http://www.sable.mcgill.ca/mclab/projects/tamer/.

[Kum14] Vineet Kumar. Mix10: Compiling matlab to x10 for high performance. Master's thesis, April 2014.

[LA04] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Run-*

time Optimization, Palo Alto, California, 2004, CGO '04, pages 75–. IEEE Computer Society, Washington, DC, USA.

[LH14] Xu Li and L. Hendren. Mc2for: A tool for automatically translating matlab to fortran 95. In *Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), 2014 Software Evolution Week - IEEE Conference on*, Feb 2014, pages 234–243.

[Mata] MathWorks. MATLAB Coder. `http://www.mathworks.com/products/matlab-coder/`.

[Matb] MathWorks. MATLAB: The Language of Technical Computing. `http://www.mathworks.com/products/matlab/`.

[Matc] Mathworks. Mex. `http://www.mathworks.com/help/matlab/ref/mex.html`.

[Matd] MathWorks. Parallel Computing Toolbox. `http://www.mathworks.com/products/parallel-computing/`.

[NBGS08] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with cuda. *Queue*, 6(2):40–53, March 2008.

[Par] Terrence Parr. ANTLR. `http://www.antlr.org/about.html`.

[QMSZ98] M.J. Quinn, A. Malishevsky, N. Seelam, and Y. Zhao. Preliminary results from a parallel matlab compiler. In *Parallel Processing Symposium, 1998. IPPS/SPDP 1998. Proceedings of the First Merged International ... and Symposium on Parallel and Distributed Processing 1998*, Mar 1998, pages 81–87.

[Rit] Dennis Ritchie. C Language. `http://cm.bell-labs.com/who/dmr/chist.html`.

[Sch75] J. T. Schwartz. Automatic data structure choice in a language of very high level. In *Proceedings of the 2Nd ACM SIGACT-SIGPLAN Symposium on*

Principles of Programming Languages, Palo Alto, California, 1975, POPL '75, pages 36–40. ACM, New York, NY, USA.

[WS81] Zvi Weiss and Harry J. Saal. Compile time syntax analysis of apl programs. *SIGAPL APL Quote Quad*, 12(1):313–320, September 1981.

[ZX] Werner Saar Zhang Xianyi, Wang Qian. OpenBLAS. `http://www.openblas.net/`.