COMPILER AND MEMORY LAYOUT IN FRESH BREEZE SYSTEM

by

Chao Yang

A thesis submitted to the Faculty of the University of Delaware in partial fulfillment of the requirements for the degree of Master of Science in Electrical and Computer Engineering

Winter 2016

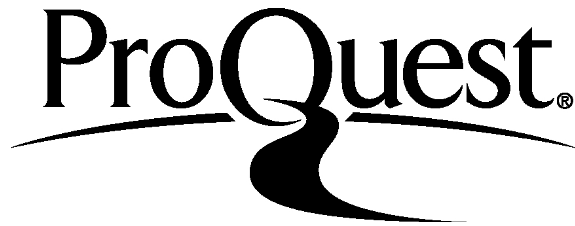
© 2016 Chao Yang
All Rights Reserved

ProQuest Number: 10055514

All rights reserved

INFORMATION TO ALL USERS
The quality of this reproduction is dependent upon the quality of the copy submitted.

In the unlikely event that the author did not send a complete manuscript
and there are missing pages, these will be noted. Also, if material had to be removed,
a note will indicate the deletion.



ProQuest 10055514

Published by ProQuest LLC (2016). Copyright of the Dissertation is held by the Author.

All rights reserved.
This work is protected against unauthorized copying under Title 17, United States Code
Microform Edition © ProQuest LLC.

ProQuest LLC.
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106 - 1346

COMPILER AND MEMORY LAYOUT IN FRESH BREEZE SYSTEM

by

Chao Yang

Approved: ______________________________

Xiaoming Li, Ph.D.
Professor in charge of thesis on behalf of the Advisory Committee

Approved: ______________________________

Kenneth E. Barner, Ph.D.
Chair of the Department of Electrical and Computer Engineering

Approved: ______________________________

Babatunde A. Ogunnaike, Ph.D.
Dean of the College of Engineering

Approved: ______________________________

Ann L. Ardis, Ph.D.
Interim Vice Provost for Graduate and Professional Education

ACKNOWLEDGMENTS

Foremost, I'd like to express my gratitude to my advisor, Dr. Xiaoming Li, who give me the research direction and insightful advice guiding me to finish this thesis.

Besides my advisor, I'd like to thank all the members in the Fresh Breeze project: Prof. Jack Dennis, Dr. Willie Lim and Mo Zhou, from the MIT Computer Science and Artificial Intelligence Laboratory (CSAIL), for sharing their wisdom and supports on developing the project; Dr. Haitao Wei for contribution of useful comments in the weekly discussion, he is under the lead of Prof. Guang R. Gao from the Computer Architecture Parallel Systems Laboratory (CAPSL) of the University of Delaware

TABLE OF CONTENTS

LIST OF TABLES vi
LIST OF FIGURES vii
ABSTRACT ix

Chapter

1 INTRODUCTION 1

1.1 Contributions 2
1.2 Outline 2

2 OVERVIEW OF FRESHBREEZE 4

2.1 Fresh Breeze Memory Model 4
2.2 Codelet Execution Model 5
2.3 FreshBreeze Processor 8

3 FRESHBREEZE COMPILER 11

3.1 Fresh Breeze Compiler Framework 11
3.2 Data Flow Graph 12
3.3 Data Flow Graph for Codelet 13
3.4 Array Representation in FreshBreeze 14

4 TRANSFORM PROCESS 17

4.1 Identify loops 18
4.2 Data Flow Graph Templates 18
4.3 Construct codelets for tree structure array 19

5 EVALUATION **22**

5.1 Linear Algebra Kernels 22

5.1.1 Dot Product 22
5.1.2 Matrix Multiplication 24
5.1.3 LU Decomposition 27

5.2 Experimental Results 29

6 MEMORY LAYOUT COMPARISON **32**

6.1 Memory Layout in Conventional Architecture 32
6.2 Memory Layout in Fresh Breeze Memory System 35
6.3 Experiments and Evaluation 37

7 CONCLUSION AND FUTURE WORK **43**

BIBLIOGRAPHY **44**

LIST OF TABLES

5.1 Data size and concurrent tasks of the benchmarks 30

LIST OF FIGURES

2.1 The fork/join relations of the sample program 8

2.2 Structure of a Fresh Breeze multi-core system 9

2.3 The details of crossbar architecture in the memory network 10

3.1 Fresh Breeze compilation flow. 11

3.2 The Data Fow Graph 12

3.3 The if conditional node of the codelet Data Flow Graph 13

3.4 The Dot Product DFG containing a While Node serves as a SubGraph Node 14

3.5 Two dimensional array represented by the tree-of-chunks structure . 16

4.1 A set of transformed codelet DFG for a reduction on a one-dimensional array 20

5.1 Codelets structure and data layout for dot product. 24

5.2 Codelets structure of matrix multiplication. 25

5.3 Transformed templates for matrix multiplication 26

5.4 Codelets structure of LU decomposition 29

5.5 Speedup normalized to one processor core. 30

6.1 A matrix in the row major order and the column major order . . . 33

6.2 Z Morton Layout 34

6.3 Tree of Chunks in Z Morton Layout 35

6.4 Matrix multiplication speedup normalized to one core in various layout 38

6.5 Auto buffer miss rate vs buffer size in various layout 39

6.6 Memory network average latency vs number of memory unit in various layout 40

6.7 The performance on Matrix Multiplication, speedup vs baseline rrr layout 41

6.8 The performance on five-point stencil, speedup vs baseline rr layout 41

ABSTRACT

In the past years, processors have evolved into multi-core, and programs also turn into multi-threading. To meet the demand of parallel processing, we face new challenges to explore the inexplicit parallelism in programs and ease the programming on large scale parallel system. The MIT Fresh Breeze system applies principles of data flow computing, leverages the write-once and chunk based memory model to eliminate the cache coherence issue in modern multi-core processor.

We develop a template based transform component in the Fresh Breeze compiler. It enables compiling funJava (subset of Java language) programs into codelet, automatically adapts the linear space memory model to the Fresh Breeze memory model. We test three common algebra function, demonstrate the Fresh Breeze system's massive concurrent execution ability and achieve the sub linear speedup performance.

In addition, through evaluations by tuning the major features, we find optimal configurations for fresh breeze processor design. Comparison on various combinations of memory layouts proves that the conventional layout optimization techniques are able to effectively apply on the Fresh Breeze system. It brings future work on the global layout optimization for a group of functions in applications.

Chapter 1

INTRODUCTION

Over the last few years, multi-core processors become to be the usual processors in computers. The parallel computing achieves better performance than the single core's through the parallelism rather than the increased clock speed. However, to achieve high performance through more scalable many-core processors, we are facing several challenges in resources utilization, cache coherency, memory consistency, programmability, etc.

In the multi-core architecture, a program is divided into multiple tasks and executed in parallel could potentially run faster than a single-core program. To realize such benefit, it requires programmers to explore the inexplicit parallelism in the program, extract code snippets into tasks, schedule task running order, etc. All the work is not trivial. For example, the imbalanced workload lowers the core utilizations and prolongs the total execution time even exceeding the single-core's running time. It is attractive to have a better approach to decompose program into tasks and coordinate them.

Furthermore, the L1 cache is privately owned by each core, and higher level(L2 or L3) cache is shared between cores. In such architecture, to maintaining consistency of the data stored in multiple caches, a cache coherent protocol is employed. In existing cache coherence protocols, the snooping protocol broadcasts a large volume of coherent messages while adding additional cores. For this reason, it limits the scalability of multi-core architect. On the other hand, the directory-based protocol brings in additional latencies, makes a trade-off for the message bandwidth. Despite many studies on cache coherence protocols, there is no satisfactory solution for this well known limitation.

Another challenge of parallel computing is writing programs with correct synchronizations. On contemporary architectures, synchronization plays the crucial role to resolve concurrent memory accessing. And it is also the difficult part for programmers who lack the knowledge of programming for parallel computing. Besides, the widely used multithreading exhibits non determinate behavior, makes it difficult to resolving the errors in programs. A better underly memory system and software interface could significantly reduce the efforts on programming and debugging.

Given these context, MIT Fresh Breeze (FrBz) project[4] builds a hardware and software co-designed tool, aiming at many-core chip architecture. Frbz architecture creates a massively scalable system, and at the same time avoids designing difficulties of the parallel programs. Frbz program execution model(PXM)[7], adapts data flow model, leverages novel chunk-based and write-once memory system, and supports fine-grained resource management. For the simplicity in the programmability, Frbz compiler provides automatically parallel programming for certain loops through FunJava, a subset of Java in the functional language style.

1.1 Contributions

This thesis makes following contributions:

- A loop analyze and transformation component, which target on Fresh Breeze codelet execution model and write-once memory system.
- Translating linear space arrays into tree-of-chunks representation for Fresh Breeze memory system .
- A Fresh Breeze memory system emulator for exploring system configurations to achieve optimal performance.
- Comparison of performance results with various memory layouts.

1.2 Outline

We begin by illustrating the overview of the Fresh Breeze memory model and programming model in chapter 2. Next we describe the Fresh Breeze compiler in chapter 3, and details of transformation component is presented in chapter 4. Chapter 5

shows how the chosen linear algebra kernels are represented by tree-of-chunks matrices in codelet and its evaluation results. In chapter 6, various performance metrics in different memory layout are presented. Lastly the paper conclusion and discussion for the future work are stated in chapter 7.

Chapter 2

OVERVIEW OF FRESHBREEZE

FreshBreeze is a scalable many-core architecture, proposes a write-once memory system organized in a single uniform address space. The write-once feature defines that each memory location is wrote only one time, but read many times. It eliminates the cache updates between cores, and avoids the bottleneck of cache coherence problem in the multi-cores system. The FrBze memory is shared globally and divided into fixed-size chunks, facilitates hardware garbage collection. The FrBz programming model adapts from dataflow model and fork/join model, provides fine-grained resource management through a simple block of instructions, called codelet.[7]

2.1 Fresh Breeze Memory Model

In the Fresh Breeze system, data and code are organized as a collection of fixed size chunks[6]. Each chunk with 128-bytes capacity, may holds 16 long or double type scalar, or 32 int or float type values. Each chunk has a 64-bit unique identifier(UID) as its handle. So, it also may hold 16 chunk handles to form a directed acyclic graph(DAG) to represent data structures. For example, a vector with 256 elements could be represented in a two-level tree of chunks, shown in figure 3.5.

In the Fresh Breeze write-once memory model, the life cycle of a chunk is depicted as follows: (1) A chunk is created by a task; (2) The producer task writes values and seals the chunk. Once the chunk is sealed, any further modification is not permitted; (3) The chunk is shared in read-only mode with other consumer tasks only after its sealing. (4) The chunk is reclaimed through the garbage collection when no task is reference to it.

The fundamental purpose of the write-once property is that the cache system would not encounter the coherency issues. The parallel tasks on the separated processing unit may read data without concern about the data is modified by other running tasks. The chunk as the basic memory unit is well suited for the efficient allocation and collection mechanisms. The reference counter and fixed size further reduce the overhead of reclaiming memory.

To create and access data structures composed of chunks, a set of memory instructions are used to as follows:

- Create - The `ChunkCreate` instruction allocates a new chunk from memory unit, returns the handle of the chunk. All the elements in the chunk are initialized as undefined.
- Write - The family of write instructions are in the form of `Write (handle, offset, data)`, write the data values to the data fields specified by the offset into the chunk identified by the handle. The argument offset could be taken from a register or a literal value.
- Read - The family of reads instructions have a similar form to the write instructions, as `destination <- Read (handle, offset)`. And it returns the value to the destination, a specified processor register.

In general, the Fresh Breeze memory model provides a virtual and global flat address environment shared between processing units without the conventional cache coherent problem. It may be extended to support various storage system like files and databases.

2.2 Codelet Execution Model

Codelet[12][10] is a simple block of instructions, it begins execution in the case that all the inputs are ready to access. In Fresh Breeze system, a single execution of a codelet is a task, which is the basic unit for scheduling.

The codelet execution model is build upon the dataflow model. Dataflow model provides crucial features for parallel computation, including find-grained synchronization, composability, and determinate execution. Especially, dataflow model explicitly

describes the data dependency, fully exploits the parallelism in a given program, provides sufficient tasks to utilize the system. Therefore, dataflow model is one of the solid foundations for building a large scale many-core system.

The codelet model[8] is a hybrid von Neumann/dataflow execution model. It inherits the features of dataflow model, adapts the fork/join model in a way similar to the parallel programming language Cilk. The codelet execution model is illustrated in figure 2.1. A master task spawns a group of worker tasks up to sixteen and then quits. Each worker task takes the arguments from the master codelet, makes individually computing contribution and then quits. Once all workers finished, a continue task is invoked, and takes the results produced by the workers. A program can be implemented by applying this schema in the hierarchy. Need to be mentioned, the codelet model differs from Cilk in that the master task quits directly without waiting worker tasks join, and there is no interaction between the master and the worker.

To support such tasking model, the Fresh Breeze system adds following tasking instructions:

- **CreateSync** instruction creates a sync chunk, a special type of memory chunk. The sync chunk serves as the join point for the worker tasks. It specifies the continuation codelet that is to be executed after workers complete their tasks. The sync chunk's handle as the return value of the CreateSync instruction is one of the arguments passed to worker tasks, it indicates the memory address for workers to update completed results.
- **TaskSpawn** instruction first creates a Task Record including worker codelet, and arguments which contain the worker ID in the range of [0..15], the sync chunk handle and dependent arguments, like TaskSpawn(Codelet, syncChunkHandle, workerID, arguments). Then the records are putted into the pending task queue of the processing unit to request the execution of subtasks in the future.
- **SyncUpdate** delivers the return result from a worker task to the position in the sync chunk specified by the worker ID provided by TaskSpawn. When all the worker tasks have completed, the specified continuation codelet is to be spawned. This process is similar to the join of thread.
- **TaskQuit** instruction means current task is finished, yields the processing unit.

A sample codelet program presents as follows and its corresponding fork/join relations is shown in Figure 2.1.

```
1  Codelet0(sync, id, args[]){
2    syncChunk0 <= SyncCreate (Codelet4) //continue to Codelet4
3    ...
4    TaskSpawn(Codelet1, syncChunk0, 0, arguments) //worker1.0
5    ...
6    TaskQuit
7  }
8  Codelet1(sync, id, args[]){
9    syncChunk1 <= SyncCreate (Codelet3) //continue to Codelet3
10   ...
11   TaskSpawn(Codelet2, syncChunk1, 0, arguments) //worker2.0
12   TaskSpawn(Codelet2, syncChunk1, 1, arguments) //worker2.1
13   ...
14   TaskQuit
15 }
16 Codelet2(sync, id, args[]){
17   ...
18   SyncUpdate(sync, id, result) // update syncChunk1
19   TaskQuit
20 }
21 Codelet3(sync, id, results[]){
22   ...
23   SyncUpdate(sync, id, result)  // update syncChunk0
24   TaskQuit
25 }
26 Codelet4(sync, id, result){
27   TaskQuit
28 }
```

In chapter 3, we illustrate how to use the memory instructions cooperated with codelet tasking instructions in constructing and accessing a vector represented by tree-of-chunks.

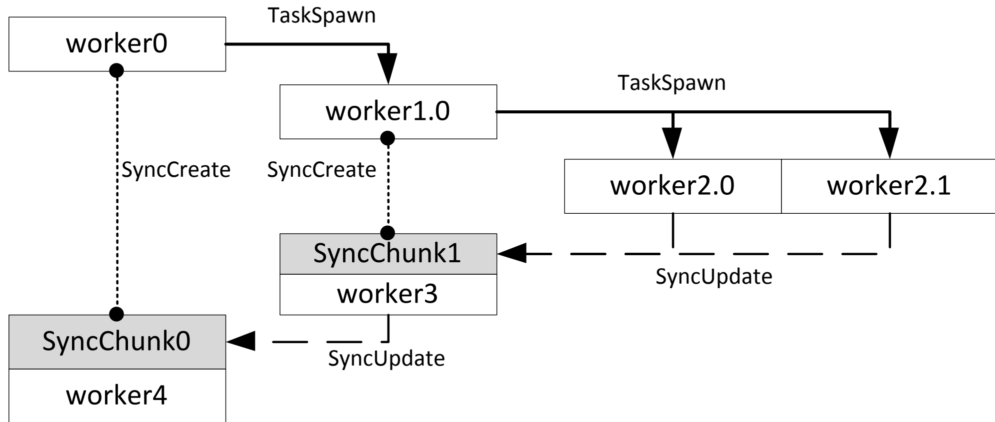


Figure 2.1: The fork/join relations of the sample program

2.3 FreshBreeze Processor

As shown in Figure 2.2, the FrBz processor is consist of a set of processing units, each unit associated with one auto buffer and one task scheduler. The memory unit is a global visible and uniformly addressable memory bank, which holds 64k chunks. Load balancer coordinates all the task schedulers, and memory network interconnects the auto buffers and the memory units.

The processing unit is a conventional RISC core, including register-to-register operations, the memory load/store instructions and codelet tasking instructions introduced in previous sections. To hide the latency of memory read instructions, processing unit equips several execution slots to support context switch between tasks. Note that when a codelet task begins execution, it occupies a slot until its termination. So, Fresh Breeze system is more suitable for programs with massive codelets that are executed in parallel and used a short running time. This codelet construction requirement is accomplished by Fresh Breeze compiler.

The auto buffer is a set of buffers holding memory chunks, with meta-data

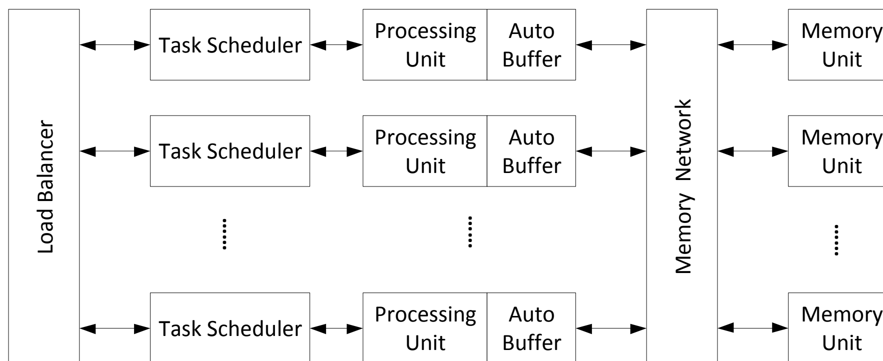


Figure 2.2: Structure of a Fresh Breeze multi-core system

indicating type and auxiliary fields for cache management. The writing police of auto buffer has similarity to the write-back cache in conventional processors. If a required chunk's handle does not existed in the auto buffer, an auto buffer miss has occurred. And then, auto buffer sends out a chunk load command to retrieve addressed chunk. When a codelet task quits, auto buffer seals all the new allocated chunks and writes them back to corresponding memory units. Note that, under the write-once rule, memory writing instruction only occurs in the processor associated auto buffer and the process of write back. Therefore, those unsealed chunks in the auto buffer wait updates and are unable to be shared between processors. In other word, there is no need to generate coherent messages between auto buffers. On the other hand, the sealed chunk stored in the memory unit, could be shared in any numbers of auto buffers, but can not be modified anymore. This is the mechanism of write-once police successfully eliminating the cache coherent problem.

The memory network serves as the bridge between auto buffers and memory units. It contains two independent paths: the forward path passes memory access command issued by processing units to memory units, the backward path returns the responses back. This network is implemented as a multi-input and multi-output crossbar switch. As detailed architecture shown in figure 2.3, a 2x2 router delivers

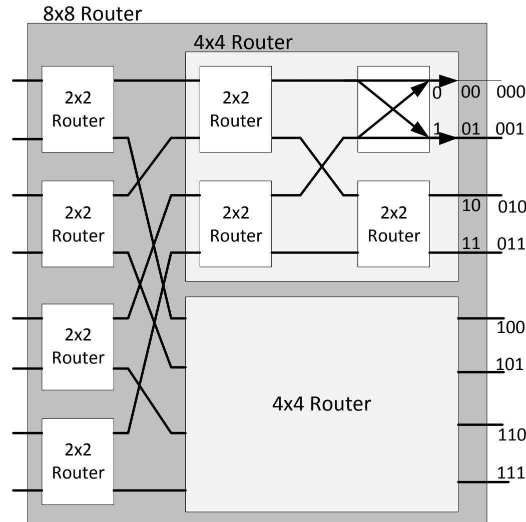


Figure 2.3: The details of crossbar architecture in the memory network

package based on 1 bit address. when two packets arrive at the same time and with the same destination address, the packet's routing priority is arbitrated in the round robin way. Clearly, the routing conflicts invoke additional latencies, the performance impact is discussed in figure 6.6.

As early mentioned, the task scheduler maintains a task queue that holds the task records waiting to be executed. The number of pending tasks are monitored by a load balancer. The load balancer implements a dynamic load balance approach, evenly redistributes the tasks from heavy loaded task queues to the other task queues with fewer tasks.

Chapter 3
FRESHBREEZE COMPILER

3.1 Fresh Breeze Compiler Framework

Fresh Breeze compiler takes Java source files as inputs and generates a list of Fresh Breeze machine instructions. Figure 3.1 shows the overall work flow of the compiler. The process starts with the standard *javac* compiler to convert Java source code to JVM bytecode. *Class File Reader* translates the stack-oriented Jave bytecode into a flat internal representation: an indexed method list and instruction lists for methods, and then it constructs a corresponding intermediate representation in the nested *data flow graph (DFG)*.

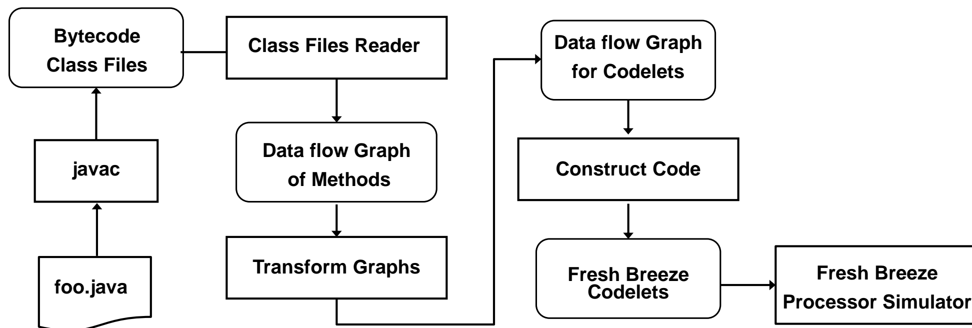


Figure 3.1: Fresh Breeze compilation flow.

The *Transform Graphs* component performs standard loop optimizations and transforms identified data flow graph into the forms suitable for the codelet model. In particularly, it analyses the loops to fully reveal the data parallelism. If a loop is eligible for parallelization, the sequential loop will be converted into a set of data flow graph fitted for Fresh Breeze parallel execution. Our work mainly focuses the

Transform component part, which are presented in section 4. It reconstructs the loops using the fork-join patten and adapts the linear space memory model into the Fresh Breeze write-once memory model.

The *Construct Code* component converts DFG for codelets into a set of Fresh Breeze codelets defined in the forms of instruction lists. The details of the *Construct Code* component part are presented in Fresh Breeze paper[5].

3.2 Data Flow Graph

In the static dataflow model, a Data Flow Graph(DFG)[3] is a directed graph composed by nodes, arcs, and tokens, shows the data dependencies between a number of functions. Figure 3.2 is an example of DFG that represents calculation of $(a - b) + (a - b) * (c + 1)$ and the intermediate result $(a - b) * (c + 1)$.

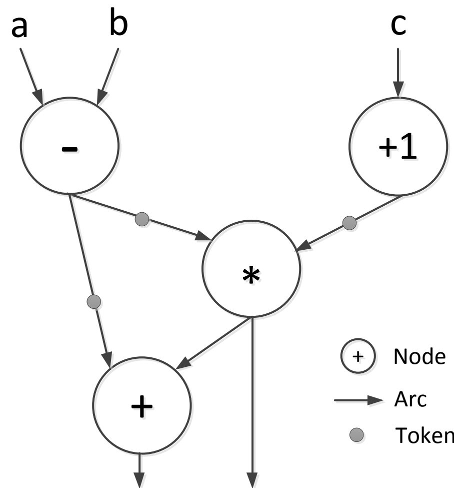


Figure 3.2: The Data Fow Graph

- **Node** is a function in DFG, may has one or more inputs and outputs. The semantics of fire rule for the node is that when all the input tokens are ready, the node consumes inputs, performs the specified operation over them, and finally produces tokens on its output ports.

- **Arc** is a directed edge, represents the data dependency between nodes, describes sources and destinations of data as well as defined and used.
- **Token** is the data passing on the arc between nodes.

The DFG represents methods (or functions) of modern structured programs without side-effects. It constraints the sequence of operator executions as well, therefore, makes the program analysis easily.

3.3 Data Flow Graph for Codelet

In the codelet model derived from dataflow model, computation is naturally modeled as a nested DFG. A DFG comprises a set of DFG nodes, and two special nodes serve as the graph's input and output ports: the source and the sink nodes. Each node has its own input ports and output ports. The ports are linked by edges, and the data "flow" along the edges from outputs to input ports.

To support the conditional branches and the function calls in programs, codelet DFG[1] is enhanced with control flow nodes, which are a group of special function nodes, including:

- **TaskApply Node** implements the TaskSpawn instruction described in Fresh Breeze memory model, and is corresponding to the method call in programs.

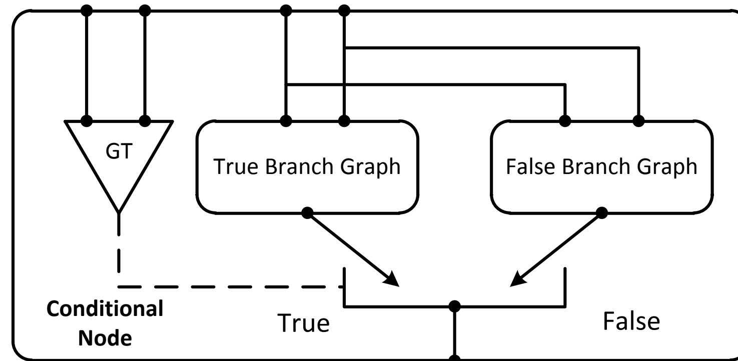


Figure 3.3: The if conditional node of the codelet Data Flow Graph

- **IF Conditional Node** in figure 3.3, contains two sub graphs as branches' body graph and one switch to select result from the true or the false branch under the control of the comparing node's result.

- **ForAll/While/Until loop Node**, shown in figure 3.4, has several special ports (red spots on the while node) called *loop ports*, which are used for passing the values of loop variables from the current iteration to the next one. In the dot production example, the while node contains a body statement: **sum** += **A**[**i**] * **B**[**i**]; its loop variables **i** and **sum** are passed from output loop ports to input loop ports for the **i+1**th iteration. When the loop condition turns to be false, loop node exits and produce the final sum results on the outputs.

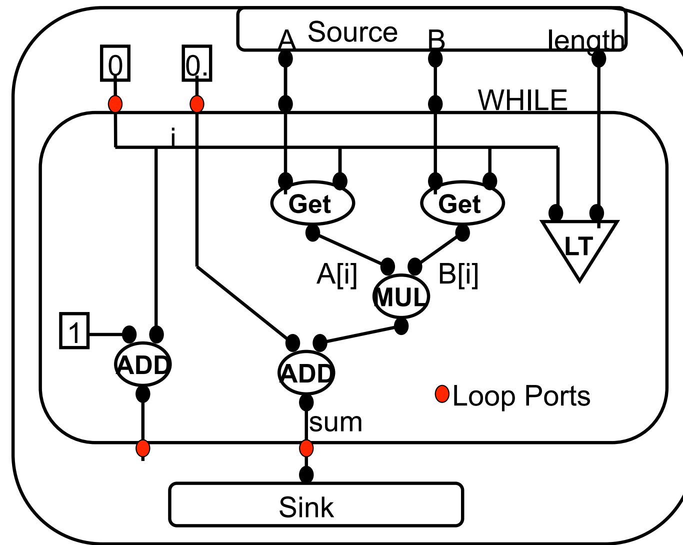


Figure 3.4: The Dot Product DFG containing a While Node serves as a SubGraph Node

- **SubGraph Node** encloses a whole DFG as a single node, is equivalent to inline a function in the conventional programming language. The previous mentioned dot product method may be treated as a SubGraph Node used in the hierarchical DFG.

3.4 Array Representation in FreshBreeze

As Fresh Breeze memory model previously described, data structures are in the form of DAG with fixed size chunks and handles. Since the chunk size of 128 bytes, each chunk can store 16 handles of 8 bytes, or 16 64-bit type data values. The data structure

of array is represented in the tree-of-chunks to naturally support operations in a divide and conquer approach. We begin with introducing the tree based one dimensional array(vector), and then extend it to the representation of multidimensional array. The data values are only stored in the leaf nodes of the tree. With the limitation of chunk size, each leaf node only holds up to 16 64-bit elements in the row-major order. Instead of carrying the data values, the non-leaf nodes hold the handles of the leaf nodes or the other non-leaf child nodes.

Examining the storage efficiency for a vector of size n, it uses $\lceil n/16 \rceil$ leaf chunks to store values, and $\lceil n/16^k \rceil$ non-leaf chunks at the `k`th tree level. Therefore, by solving the recurrence relation $T(n) = T(n/16) + O(n)$, the total memory usage is O(n). For a typical scenario, array built in a tree of k depth uses approximate $16^{(k-1)}$ extra chunks for storing handles, for example, vector size of 4096 uses 256 chunks for data and only 17 extra chunks for handles. To fully utilize the memory, empty chunks are not created in the tree structure. For instance, an array with 17 data elements is expressed in a two-level tree with only 3 memory chunks: 2 leaf chunks keep data values and 1 root chunk contains the handles of leaf chunks.

Analyzing the runtime of random accessing individual data element, it requires traversing the tree from the root through the depth of tree, which is $O(log_{16}(n))$. This tree traversing cost increases relative slow comparing with the array size. For a typical instance, the size of an array expands by 16 times, the depth just increases by 1. The tree-of-chunks array is not designed for such random individual reading pattern, but for the parallel processing and scalability. Especially in the for-each pattern, the tree structure is naturally divided into sub trees and data chunks are accessed in parallel. In such case, processing the entire array takes O(n) runtime, equal to the number of elements in the tree.

Multidimensional array is organized in the “tree-of-trees” structure. Each tree serves as one of the dimensions in the array. The leaf node of the `i`th dimension contains the handles pointing to the root of the `i+1`th dimension trees. Similar to the case of one dimensional array, data values are only stored in the leaf nodes of the last

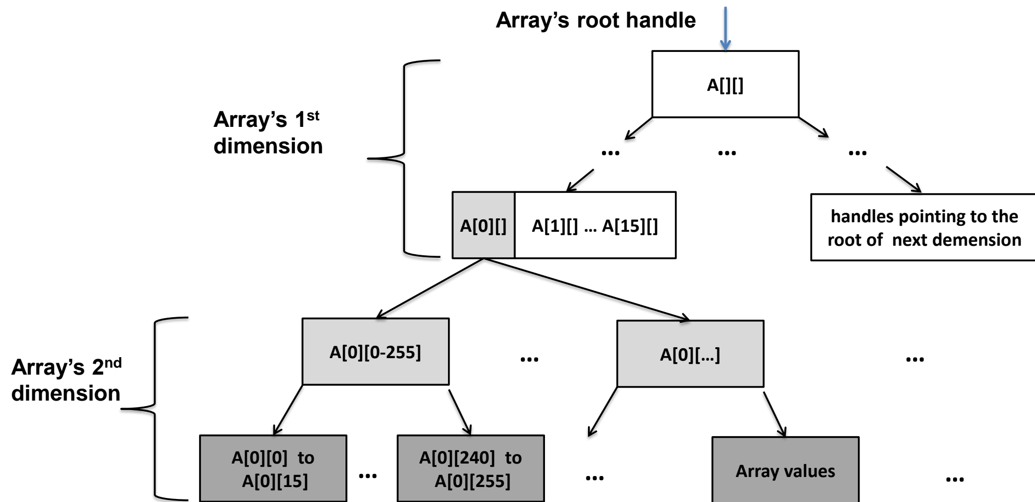


Figure 3.5: Two dimensional array represented by the tree-of-chunks structure

dimension. Examining the space and time complexity of the multidimensional array, the storage of n^d elements in a d dimensional array requires the production of spaces for every dimensions, therefore, the total usage is still $O(n^d)$. Random accessing requires the traversing the summation of tree depths in every dimension, which is $O(d \cdot log_{16}(n))$ in total.

Chapter 4

TRANSFORM PROCESS

The major functions of transform component are (1) adapting accessing on traditional linear arrays into tree represented arrays in fresh breeze memory model; (2) transforming sequential loops with data parallelism into parallel execution codelets in the form of the codelet data flow graph.

The challenges in transformation is how to modify linear memory model programs to conform to the write-once rule and the tree-based array representation. To further explanation, there are 3 types of memory operations: read, write and update. For the array read operations, direct address access is replaced by tree traversing. For creating a new array, which are all write operations, restricted by a chunk is immutable once it is created by a codelet, we have to build the tree structured array in the bottom-up approach. Updating elements in arrays is decomposed into two steps: firstly read the array, and then create a new array where only the specified elements are changes, and the other unmodified values are copied from the original array.

For the loop transformation challenge, by analyzing loops in each method DFG, loop nodes are classified as *array constructions or reductions or irregular loops.* The transform component recognizes former two forms of data parallel loops that benefit from Fresh Breeze architecture directly: (1) a reduction loop in which a reduction operator is applied to values provided by the loop body to generate a final scalar result; (2) an array construction loop in which each value produced by the loop body becomes an element of an array which is the result of loop execution. The irregular loops still get benefits if it could partly conform with previous two forms. Transformation component following the methodology: first analysis and plan the code then construct

it. It includes two steps: identifying data parallel loops and building an attribute tree; and the loops replacement through template based DFG.

4.1 Identify loops

Since the program described in the nested DFG, the analysis is clearly carried out recursively. The loop nodes DfgUntil or DfgWhile that are amenable to data parallel, require no dependencies among executions of the loop body. In addition, to be fitted into the codelet model, loop nodes have to meet following restrictions:

- No more than 2 loop ports: one for the loop induction variable and the other for the optional result provided by the loop body.
- When A loop carries reduction operand to produce a single loop output result, it requires that each iteration of the loop defines a scalar value, and the set of scalars is combined using a commutative-associative operator to produce a final loop result; and there is no array write operation in the loop body.
- When the single output is a chunk's handle, the loop expresses an array construction, only if each iteration of the loop body defines an element of the result chunk without references to the same array name of the loop output; and the element index is exactly the loop induction variable and increased by one in every iteration.

Loop attributes used for the transformation recorded in the attribute tree, includes the loop type which is among reduction, array construction and irregular loop; the reference of the loop induction variable; and an list holding the references of array, which are needed to be traversed as independent chunk for parallel processing.

4.2 Data Flow Graph Templates

The code of loops have relative fixed structures and similar functions, thus they are transformed into data flow graphs with similar structures. Observing these graphs, they have very common structures and repeat many times when transforming. Therefore, we extract this common DFG structures as *templates* with configurable parameters and be reused to describe similar data flow graphs instead of repeating construct. Following this method, the main ideal of transforming is replacing the loop

nodes in data flow graph with parameter configurable DFG templates. The predefined templates covers five types of graphs:

- **DfgVectorForAll** is the top level template for the loop transformation, invokes a set of templates according to the loop analysis information and recursively calling itself for nested loop transformations.
- **DfgTraverseVectors** traverses tree-structured arrays recursively. It may spawns up to 16 worker tasks, each performs DfgTraverseVectors itself on a segment of array. This process continues recursively until the last non-leaf level, at which point a set of DfgCompute codelet is spawned for accessing or generating the data chunks at the leaf level of the tree.
- **DfgCompute** executes computations in the original loop body. For array construction type of loop, it generates the data chunks, which are the leaf nodes in the tree, and returns the handles as results to the template DfgVectorDone; for the reduction loop, it applies reduce operations over the input data chunks and returns a reducing value to the template DfgReduceDone which performs the rest of reduction computation.
- **DfgReduceDone** performs a reduction operation over the input data, when the loop is a reducing loop. It collects the reduction values from a group of DfgCompute or lower DfgReduceDone tasks, and returns further reduction value of this group to the upper level of tree's sync chunk.
- **DfgVectorDone** is invoked by DfgVectorForAll as the continue codelet of DfgTraverseVectors or DfgCompute, when the loop type is the array construction. It stores the handles from lower level workers into a chunk as the parent chunk of those, and updating this chunk's handle, as the child chunk in the tree, to its parent codelet.
- **DfgTraverseDepth** is a functional sub graph, using in the DfgForAllCompute to calculate the corresponding depth of the tree from a given array length.
- **DfgDummyJob** is a helper functional template, pairing with DfgVectorForAll, it simply returns the final results what it gets from the input to the continue task;

4.3 Construct codelets for tree structure array

As described in section 4.2, a similar set of DFG templates helps constructing both array construction and reduction loops. According to records in the loop attribute tree, the transformation component selects suitable loops for codelet memory model,

replaces these loops with predefined templates started from the most inner level, due to the same reason for construction an array in the bottem-up way.

Figure 4.1 shows the codelet calling relation for a one-dimensional reduction loop. The codelet on the left is the invoked worker task, and its right codelet is the paired continue task. The arrow represents the task execution order as well as the data flow. The original DFG, shown in figure 3.4, only has one single codelet applying reduction over the whole vector, and its execution is limited in the sequential order. On the contrary, the transformed DFG is a set of codelets and each of them processes a chunk independently, details in section 5.1.1. Therefore, the Fresh Breeze system easily executes each codelet in parallel.

To build the tree structured for vectors, the used templates are almost the same as those of the reduction operation, except the `DfgReduceDone`, which is substituted by `DfgVectorDone`. To create the inner nodes of the tree, `DfgVectorDone` collects a group of handles from the data chunks, and updates them into parent's sync chunk recursively until reaching the root sync chunk.

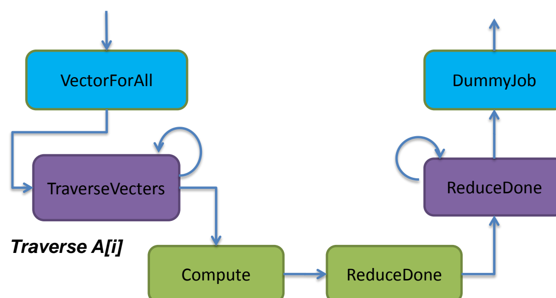


Figure 4.1: A set of transformed codelet DFG for a reduction on a one-dimensional array

In summary, the transform component analyzes the DFG of Java methods, identifies suitable loops for parallel codelet and converts those DFG of loops into a set of codelet DFGs. It automatically determines whether a given loop code is a data parallel reduction or an array construction. Then it creates the corresponding pattern of codelets accordingly. Currently, the transformation of compiler only supports loops

that can be represented as parallel *for* loops with fixed limits and a constant loop stride, similar to OpenMP. To adapt to the tree-structured array and write-onc memory model, the transformed codelets have the similar pattern to the structure of the dot product example in section 5.1.1, that is consist of “Traverse”, “Compute” and “Done” codelets. This common codelet structure serves as a template with configurable parameters, which is applied on all the codelet feasible loops. In the following chapter, we sketch the codelet DFG of three wildly used linear algebra algorithm that conform the patterns for the data parallel loops.

Chapter 5

EVALUATION

5.1 Linear Algebra Kernels

In this section, we demonstrate compiler's transformation results through three linear algebra kernels: Dot Product, Matrix Multiplication and LU Decomposition, expressed by codelet DFGs and tree-based memory model discussed above.

5.1.1 Dot Product

The dot product in sequential program is depicted as follows. It is a simple routine prefect fitted for fresh breeze system, including an array read and an array construct. The traditional approach for parallelizing a dot product computation is to divide its main loop into small parts. Each part is then executed as a separate task, which performs a small part of dot product and then does a sum reduction with the other tasks to produce the final result.

```
1   void main(int length) {
2       long[] A = BuildVector(length, 17);
3       long[] B = BuildVector(length, 19);
4       long sum = dotProd(A, B, length);
5   }
6
7   long[] BuildVector(int length, long mod) {
8     long[] vec = new long[length];
9     for (int k = 0; k < length; k++) {
10      vec[i] = k % mod;
11    }
12    return vec;
13  }
```

```
14
15    long dotProd(long[] A, long[] B, int length) {
16      long sum = 0;
17      for (int k = 0; k < length; k++) {
18        sum += A[k] * B[k];
19      }
20      return sum;
21    }
```

Figure 5.1 illustrates the principle of translating the dot product code into codelets of the tree-based memory model. First, the vectors need to be converted to tree-based memory chunks. In the current system, a memory chunk holds 16 long values. Hence vectors are divided into 16-element segments and organized as a tree structure – the leaf chunks hold the actual data values, while non-leaf chunks store handles that point to other chunks.

As shown in the example in Section 4.3, we need three codelets to construct a tree-of-chunks: A master codelet *DfgVectorForAll* checks the depth of the tree that needs to be built, if it is not the leaf node, then it recursively spawns *DfgTraverseVectors* codelets. At the leaf level, a work codelet *DfgCompute* is spawned to allocate data chunks (typically 16 elements per chunk, possibly less than 16 if it is the final chunk) and returns the handle to the Sync Chunk. A continuation codelet *DfgVectorDone* returns the handle of the Sync Chunk to the next (upper) level's Sync Chunk and this continues until the uppermost (root) level is reached.

After vectors **A** and **B** are constructed, the computation dotProd method follows. In this step, a master codelet *Traverse Vector* is executed. It takes the roots of two tree-of-chunks **A** and **B** as inputs. It then checks the depth of the tree to see if it is a leaf node. If not, it recursively spawns *DfgTraverseVectors* codelets taking the handles of the lower level as inputs. At the leaf level, a worker codelet *DfgCompute* is spawned to compute the dot product of 16 elements and return the result *sum* to the sync chunk. The continuation codelet *Reduce* adds all results in the sync chunk that are filled by lower level *DfgCompute/DfgReduceDone* codelets and returns the handle

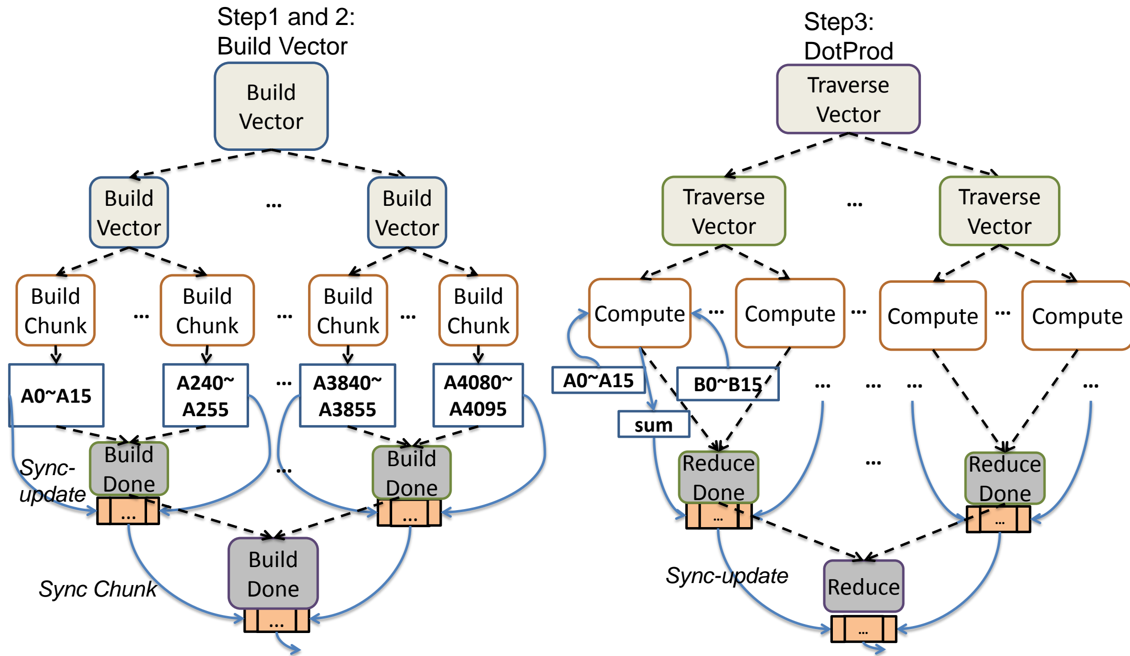


Figure 5.1: Codelets structure and data layout for dot product.

of the sync chunk at the next (upper) level sync Chunk and so on until the root level sync Chunk which holds the final result is reached.

5.1.2 Matrix Multiplication

The Matrix multiplication is an important linear algebra algorithm and a crucial part of the scientific applications. It is an ideal choice for Fresh Breeze system to study. The Matrix Multiplication is a classic embarrassingly parallel routine shown below, which is another example to introduce how 2-D matrix expressed in Fresh Breeze system. A vector is represented as a "tree-of-chunks" in which leaf nodes hold the values. A 2-D array can be viewed as a vector of vectors that is constructed by replacing the values in the leaf nodes with the root handles of vectors). In general,a multidimensional array can be organized as a "tree-of-trees" structure, where each tree represents one dimension of the array. The leaf node of the first dimension contains

the handles pointing to the root of the tree which representing the second dimension, and so forth. Data values are only kept in the leaf node of the last dimension.

```
1   double[][] MatrixMult (double[][] A, double[][] B,
2                          int nr, int nc, int nv) {
3     double[][] product = new Double[[nr][nc];
4     for (int i = 0; i < nr; i++ ) {
5       for (int j = 0; j < nc; j++ ) {
6         double[][] sum = 0.;
7         for (int k = 0; k < nv; k++ ) {
8           sum = sum + A[i][k] * B[k][j];
9         }
10        product[i][j] = sum;
11      }
12    }
13    return product;
14  }
```

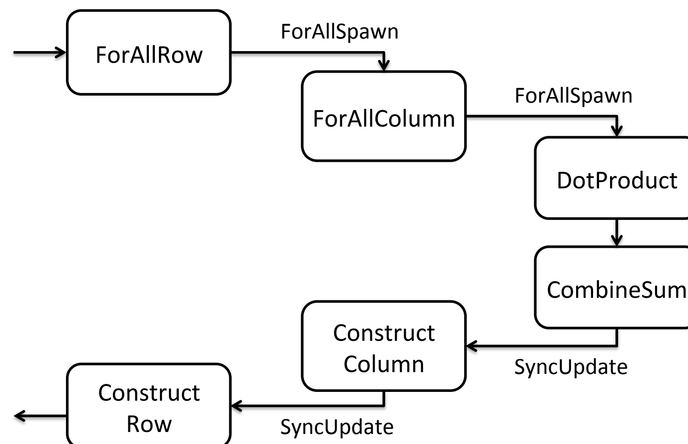


Figure 5.2: Codelets structure of matrix multiplication.

Matrix multiplication **C=A×B** follows the same two steps as in dot product: first construct the data chunks for the matrices and then perform the computation. Matrix multiplication is a three-level nested loop with dot product as the most inner

loop, and illustrated by three groups of the DFG templates in Figure 5.2. We assume that matrix B is already transposed to agree with the dimension of A. Therefore, the two matrices have the same structure when represented by trees: row first and column second. Each row vector is represented as a subtree where each leaf node stores 16 elements of the row. Since the two steps use almost the same codelet structures, we will focus on the computation step here. As figure 5.2 illustrates, the program first spawns *ForAllRow* codelets to traverse all the rows of matrix **A**. Each of the spawned codelet takes a subtree representing one row of matrix **A** as input. For each row, the *ForAllRow* codelet spawns a *ForAllColumn* codelet to traverse all the columns of matrix **B**, where each spawned codelet takes the subtree of a column of matrix **B** as input. The *DotProduct* codelet takes in one row and one column, follows the codelet structure in Figure 5.1 to compute the result of dot product. The *CombineSum* codelet returns the result value. The *Construct Column* codelet collects the results of the dot products to construct a row vector of matrix **C**. Then the *Construct Row* codelet collects all handles of rows to build the matrix **C** as the final result.

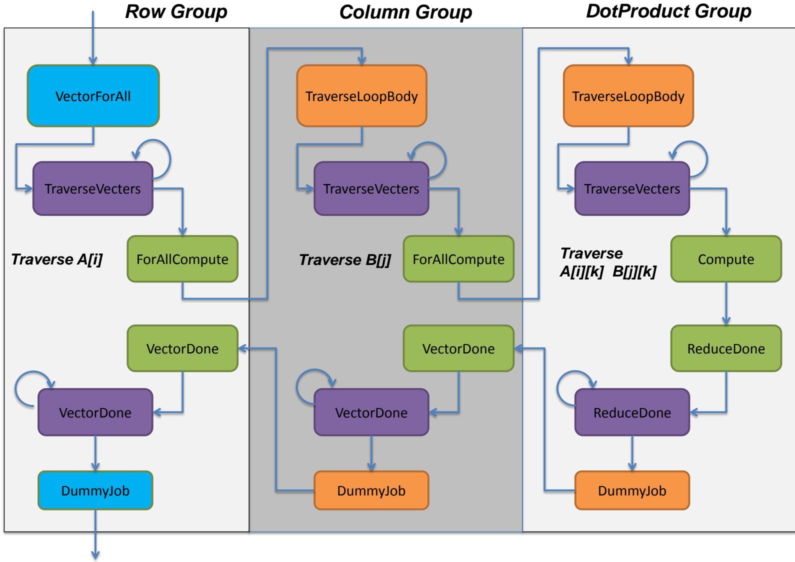


Figure 5.3: Transformed templates for matrix multiplication

Figure 5.3 illustrates the details for three groups of Matrix Multiplication DFG templates, as well as a multidimensional array construction and nested loop transformation. Each group is an vector construction. Although the basic traveling templates is same as the one dimensional case, shown in figure 4.1, addition templates ForAll-Compute and TraverseLoopBody is introduced as the tree traversing adapter between dimensions. The DfgForAllCompute substitutes the DfgCompute in the vector example, spawns a set of tasks DfgTraverseLoopBody, which is similar to DfgVectorForAll to perform vector traversing on the next dimension.

5.1.3 LU Decomposition

LU Decomposition (LUD) is a major part of linear system solvers widely used in scientific and engineering applications. It decomposes a square matrix **A** into a lower triangular matrix **L** and an upper triangular matrix **U**, so that **A**=**L*****U**. The following shows a LUD function with **L** and **U** stored in the same matrix, written in empirical programming language with side effect on array A.

```
1   //LU Decomposition with side effect on array A
2   void LUD(long[][] A, int len){
3     for (int k=0;i<len;k++){
4       for(int i=k+1; i<len; i++){
5         long t=A[i][k]/A[k][k];
6         for(int j=k+1; j<len; j++){
7           A[i][j]=A[i][j] - t*A[k][j];}
8       }
9     }
10  }
```

To handle this type of sequential loops, which has a same name array but different index in read and write operations, we require the program represent in the style of functional language without side-effect. Clearly, we find that the LUD has a barrier synchronization after each row, so it is re-factored as following code:

```
1   //LU Decomposition without side effect through recursion
```

```
2    long[][] LUD(long[][] A, int len) {
3        return LUD_Step(A, 0, len);
4    }
5    long[][] LUD_Step(long[][] A, int k, int len) {
6        if (k < len) {
7            long[][] B = new long[len][len];
8            for (int i = 0; i < len; i++) {
9                long t = A[i][k] / A[k][k];
10               for (int j = 0; j < len; j++) {
11                   if (i > k && j > k) {
12                       B[i][j] = A[i][j] - t * A[k][j];
13                   } else {
14                       B[i][j] = A[i][j];
15                   }
16               }
17           }
18           return LUD_Step(B, k + 1, len);
19       } else {
20           return A;
21       }
22   }
```

The recursive function *LUD_Step* is equivalent to a loop with k iterating from 0 to *len-1*. With matrix **A**'s loop carried dependence, each recursive call of *LUD_Step* updates a part of the matrix. This recursive call is composed of a set of codelets *LUD_Step k* shown in figure 5.4, which recursively call themselves until reaching the loop bound. In this group of codelets, *LUD k-check Continue* makes the recursive calls by increasing the value of k and spawning *LUD k-check* of the next recursion. The codelet *LUD k-check* checks recursion termination by comparing the value of k against the loop bound *len*. If k has not reached the loop bound, the codelet *LUD Compute* is spawned; otherwise, the *LUD_Step k* is terminated while the continuation codelet is spawned. Abiding by the write-once memory policy, the loops cannot modify the values of matrix **A**. Therefore, at the kth recursion, the loop-i and the loop-j construct

a new matrix **B** for the *k+1*th recursion by *LUD Compute* and *LUD Compute Done*. During this construction, the codelet *LUD Compute* updates values corresponding to the *k*th iteration and copies values that are not modified into the newly constructed matrix **B**.

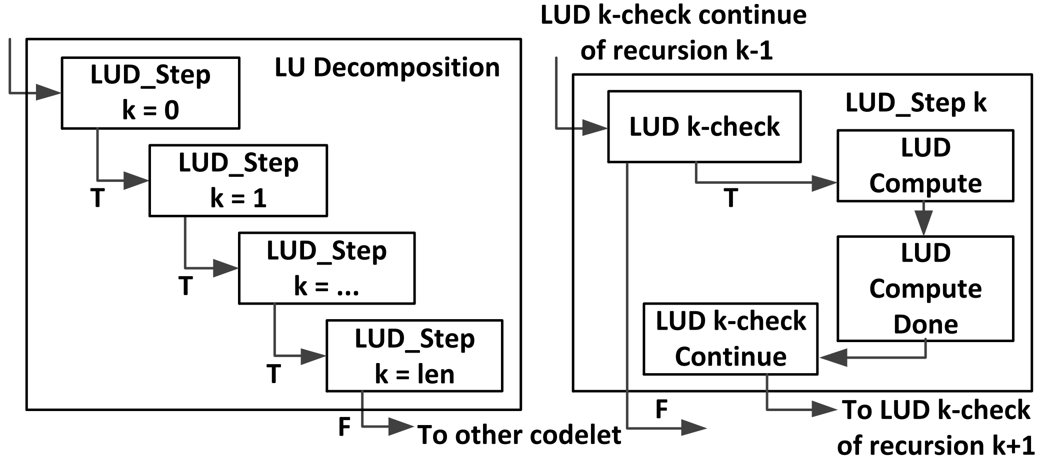


Figure 5.4: Codelets structure of LU decomposition

5.2 Experimental Results

We evaluates the examples dot Product, Matrix Multiplication, LU decomposition discussed in previous sections on Fresh Breeze processor simulator. The results are reported in this section.

Table 5.1 shows the input data size and number of concurrent computation tasks of the benchmarks. The concurrent computation task is the smallest computation codelet at the leaf node (data chunk) level. Since the current test environment of the simulator is limited to 20 bits of address space and with garbage collection not available yet, we have to limit the input data size so as not to run out of memory and yet have a large enough number of concurrent computation tasks to evaluate our system.

Table 5.1: Data size and concurrent tasks of the benchmarks

Benchmark	Data size	Concurrent computation tasks
Dot Product	16^5 (vector)	2^{16}
Matrix Multiplication	128x128 (matrix)	2^{17}
LU Decomposition	128x128 (matrix)	2^{10}/iter 2^{17} in total

Figure 5.5 shows that near linear speedup is achieved for all the benchmarks with the target system size growing from 1 to 64 cores. For the Dot Product, Matrix Multiplication and LU Decomposition the speedup for 64 cores is 29.51x, 28.32x and 22.62x, respectively. This because there are enough codelets in each of the three benchmarks to fully utilize the system.

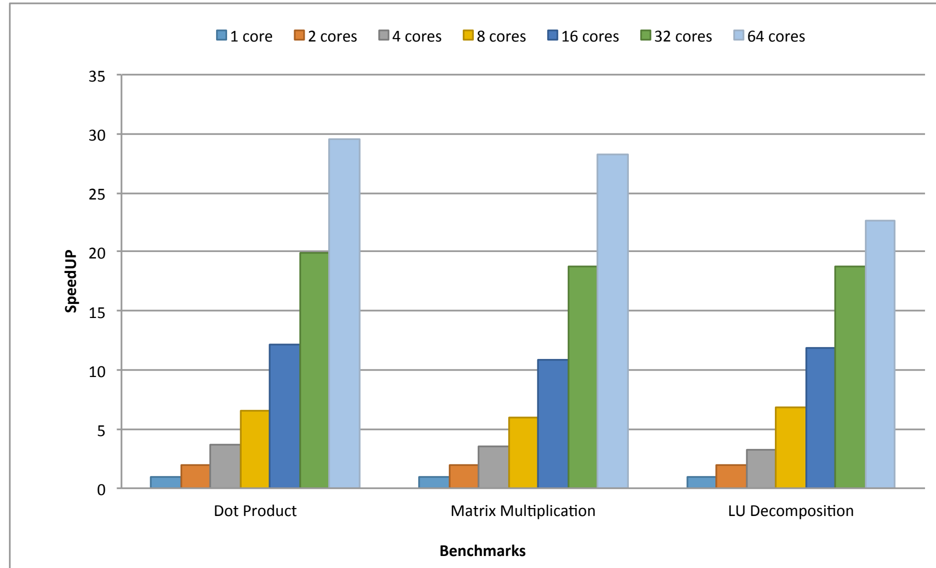


Figure 5.5: Speedup normalized to one processor core.

In summary, the transformation component successfully bridges the linear space programs into Fresh Breeze system, fully explores the parallelism among the loops, achieves a sub-linear speed-up over various programs. It provide a useful tool making

an easy way to port conventional programs into the codelet model and write-once memory system.

Chapter 6

MEMORY LAYOUT COMPARISON

The memory layout maps the multidimensional array to the one dimensional linear space preserving locality of data points. The layout is an important factor for the cache performance in a given program, since accessing array elements in contiguous pattern is much faster than the pattern which are not.

In Fresh Breeze system, the auto buffer described in section 2.3 is the cache for chunks. The tree-of-chunks array is a hierarchical and recursive representation. The memory layout may enhance the program performance of the Fresh Breeze system in the similar way. In this chapter, we present two algorithms of matrix multiplication and five-point stencil in various combinations of layouts.

6.1 Memory Layout in Conventional Architecture

The two dimensional matrices are naturally stored in two formats in the linear space memory model. One is row major order of matrix, there the consecutive elements of the rows in the array are contiguous in memory; the other is column major the consecutive elements of the columns are contiguous. Supposing a matrix with m rows, and n columns, the element at the ith row and the jth column. The mapping function of row major matrix:

$$f(i, j) = i * n + j; \tag{6.1}$$

For a column major layout:

$$f(i, j) = j * m + i; \tag{6.2}$$

Figure 6.1: A matrix in the row major order and the column major order

In both cases, accessing elements in the direction perpendicular to the data layout causes large amount of cache misses, and introduces long memory reading latencies. The example of the standard matrix multiplication routine in section 5.1.2, works poorly in both data layouts. To address such performance issue, the spacing filling curves[2] have been explored for multidimensional matrix layouts in the conventional memory system. The main idea of space filling curve is lessen the distance between the adjacent matrix element in one specific direction by recursively dividing matrix into tiles and tiles are placed in a non-linear pattern. Although the Hilbert curve has a better order-preserving behavior among many other alternative choices of filling curves, the calculations of Hilbert curve involve complex rotations, which are much more complicated and significant overhead. Hence, we choose the Z-morton layout[9] which is a basic implementation of space filling curve in matrix.

The Z-morton index value of a point in matrix is calculated by interleaving the binary representations of its coordinate values. For the two dimensional matrix, the integer coordination is by following *zMorton* function. Note that, this index translation takes 26 operations, it is an acceptable overhead for a memory access, which could be hide by multi-threading.

```
1  static final int[] B = new int[]{ 0x55555555, 0x33333333, 0x0F0F0F0F, 0
     x00FF00FF };
2  static final int[] S = new int[]{ 1, 2, 4, 8 };
```

```
3   private static int zMorton(int x, int y) {
4     x = (x | (x << S[3])) & B[3];
5     x = (x | (x << S[2])) & B[2];
6     x = (x | (x << S[1])) & B[1];
7     x = (x | (x << S[0])) & B[0];
8     y = (y | (y << S[3])) & B[3];
9     y = (y | (y << S[2])) & B[2];
10    y = (y | (y << S[1])) & B[1];
11    y = (y | (y << S[0])) & B[0];
12    return y | (x << 1);
13  }
```

0	1	4	5	16	17	20	21
2	3	6	7	18	19	22	23
8	9	12	13	24	25	28	29
10	11	14	15	26	27	30	31
32	33	36	37	48	49	52	53
34	35	38	39	50	51	54	55
40	41	44	45	56	57	60	61
42	43	46	47	58	59	62	63

Figure 6.2: Z Morton Layout

The square shape tiles is the trade-off layout between row major and column major. The distance between two adjacent elements depends on the tile size. The typical distances are within a tile size, or under an upper-level tile size. In the worst case, the maximum distance of adjacent is 1/4 of the size of matrix and it occurred at reading the element located at the boundary of a tile, and the next element is in another tile, for example on the 5th column, reading 48 immediately after reading 26.

6.2 Memory Layout in Fresh Breeze Memory System

The two dimensional array in tree of chunks shown in section 3.4, has poor performance in the column contiguous accessing. The two adjacent elements in one column requires to traverse from the root chunk, only the chunks in the first dimension may have cache hit in the auto buffer, but the chunks in the second dimension always makes cache misses. This behaviors cause large overhead in memory load.

For clear and simplicity to illustrate the tree of chunks in z morton, we use the quadrant tree instead of the Fresh Breeze tree holding 16 elements. From figure 6.3, we see that the tiles are organized in an hierarchical layout. The bottom tiles c0 to c15 are 2x2 blocks in Z morton, the upper level (the blue chunks) is consist of 2x2 bottom chunks, and the whole matrix is the group of 2x2 blue chunks. In each level, the elements' ordering all follows z morton. So, this structure is created recursively through bottom up approach, same as the vector construction, but with additional z index calculation.

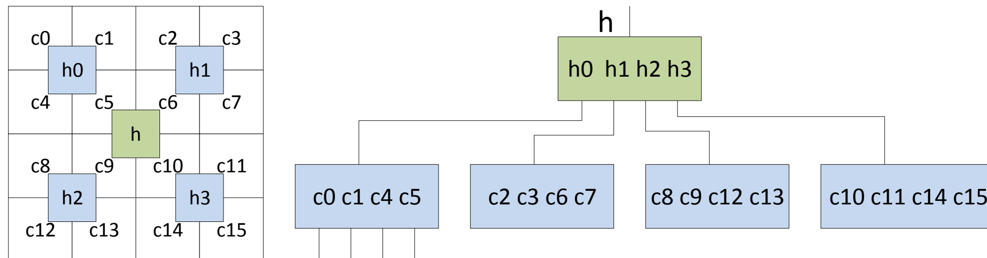


Figure 6.3: Tree of Chunks in Z Morton Layout

Following is the program of construction a tree in the recursive way. The line 13 applies a user customize function to calculate the results for the ith elements in z morton ordering. The line 20 is adding codelet into pending task queue, so the execution order for building the tree is in the breath first search way.

```
1   public void parallelConstruct(Function<Integer, Number> function) {
```

```
2     this.root = ChunkPool.allocate();
3     Task t = new Task(root, function, depth, 0);
4     tq.offer(t);
5   }
6   public static void parallelConstruct(Chunk root,
7       Function<Integer, Number> function, int traverseDepth,
8       int indexOffset) {
9     if (traverseDepth <= 1) {
10      for (int i = 0; i < root.size; i++) {
11        Integer treeIndex = indexOffset + i;
12        root.setNumber(i, function.apply(treeIndex));
13      }
14    } else {
15      for (int i = 0; i < root.size; i++) {
16        Chunk h = ChunkPool.allocate();
17        root.setHandle(i, h);
18        Task t = new Task(h, function, traverseDepth - 1,
19                          h.capacity * (indexOffset + i));
20        tq.offer(t); // call parallelTraverse2D recursively
21      }
22    }
23  }
```

From the software perspective, the memory layout directly impacts the number of auto buffer misses, also the number of memory loads, which dominates the performance of memory system. From the processor architecture view, each core supports multiple execution slots, a multi-threading approach in the codelet model, efficiently overlaps the computation time with the memory access latency. With increasing the number of cores to n, the latency of interconnect network is increased in the order of log n, not be neglected. Besides the port-to-port latency of the memory network, there are latencies invoked by the memory unit conflicts need to be considered. When multiple cores sent out memory requests to the same memory unit with different addresses, the memory unit can not serve them all at the same time. So the requests enqueued, the

conflict latency is the waiting time of queue length. [11]

6.3 Experiments and Evaluation

Based on the analysis in previous section, we simulate kernel access patterns in the linear algebra library. The test results reveal the characteristic features of Fresh Breeze memory system. To evaluate the memory system performance, we choose programs with a few computations but frequent memory accessing, assuming the time of computation part is much less than the time of memory access latency, and does not count into evaluation. The following code snippets are the user customized functions in the tree construction program, which implement matrix multiplication and five-point stencil computations. Both of the code have the intensive memory readings but simple operations.

```
1  // Matrix Multiplication
2  Function<Integer, Number> mm = (i) -> {
3    int[] dims = t2.getDimsIndex(i);
4    long sum = 0;
5    for (int k = 0; k < len; k++) {
6      sum += t0.getLong(dims[0], k) *
7           t1.getLong(k, dims[1]) ;
8    }
9    return sum;
10 };
```

```
1    // stencil
2    Function<Integer, Number> stencil = (i) -> {
3      int[] dims = t2.getDimsIndex(i);
4      long avg = 0;
5      int row = dims[0];
6      int col = dims[1];
7      avg = t1.getLong(row, col);
8      avg += row - 1 >= 0 ? t1.getLong(row - 1, col) : 0;
9      avg += row + 1 < len ? t1.getLong(row + 1, col) : 0;
10     avg += col - 1 >= 0 ? t1.getLong(row, col - 1) : 0;
```

```
11        avg += col + 1 < len ? t1.getLong(row, col + 1) : 0;
12        avg /= 5;
13        return avg;
14      };
```

The memory network is configured with 6 cycles for the latency, and the memory unit consumes one request per cycle. In the figures of evaluations, we use following notations: row major layout is denoted by "r", column major layout by "c" and z morton layout by "z". For the matrix multiplication, first 2 characters indicate the input matrices' layout, and the last letter means the memory layout of the created matrix. For example, "rcr" means a row major matrix times with a column major matrix providing the result in the row major layout. We explored the parameters of the fresh breeze processor, while analyzing the memory layouts.

From figure 6.4, as we expected, the memory layout does not show the difference of concern on the scalability of Fresh Breeze system while increasing the number of cores.

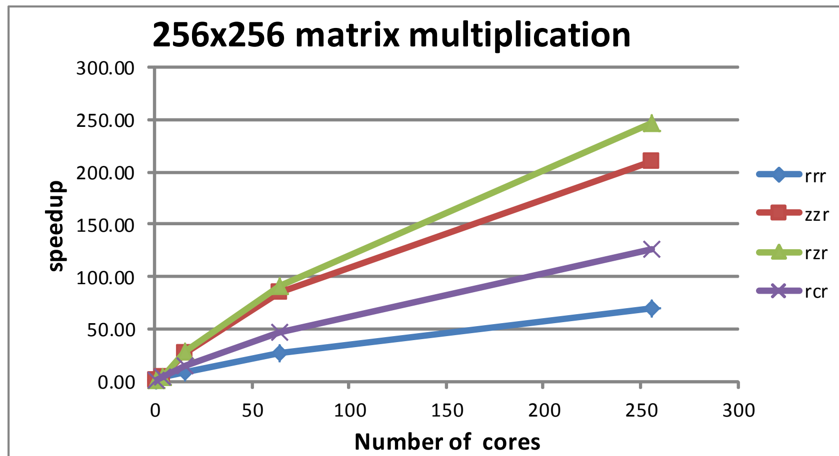


Figure 6.4: Matrix multiplication speedup normalized to one core in various layout

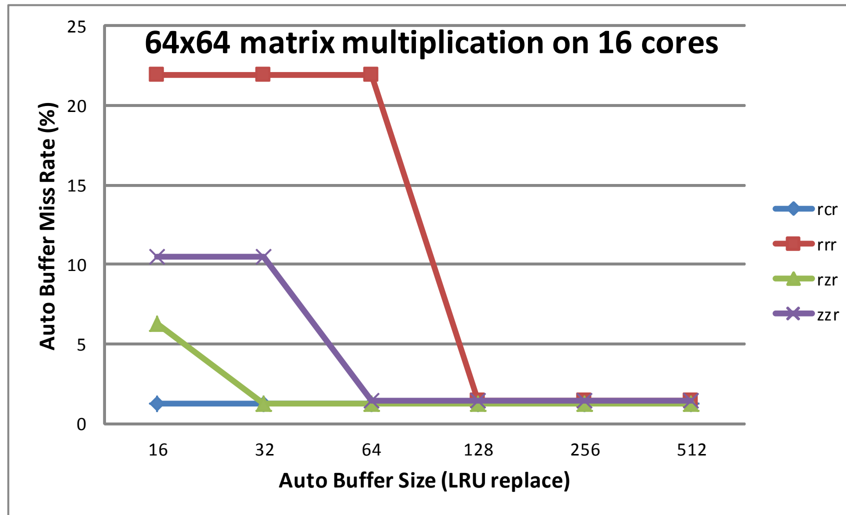


Figure 6.5: Auto buffer miss rate vs buffer size in various layout

Figure 6.5 shows that the miss rate keeps at a stable value under a limited size and reduced to optimal once auto buffer size reach at a certain size. Hence, the memory layout has a remarkable influences on the program performance, not the auto buffer size. As well known, the standard matrix multiplication favors a row major matrix times a column major matrix. In fresh breeze processor, the capacity auto buffer is 32 chunks, each chunk is 128 bytes, 4k bytes in total. Although auto buffer has a much smaller size than the 32KB L1 cache in the conventional processor, it efficiently reuses the root and inner chunks of the tree. In the typical codelet, the auto buffer may hold up to 5 matrices or vectors, each size is 4096x4096 represented in a 6 level tree.

The average latency of memory network is shown in figure 6.6. As we can see that the latency in every layout coincidently gets close to the optimal port-to-port network latency, when the number of memory unit is equal or more than the number of simultaneous executing codelets. In Fresh Breeze processor, one memory unit is binding with one processing core. Therefore, the number of memory units is same as

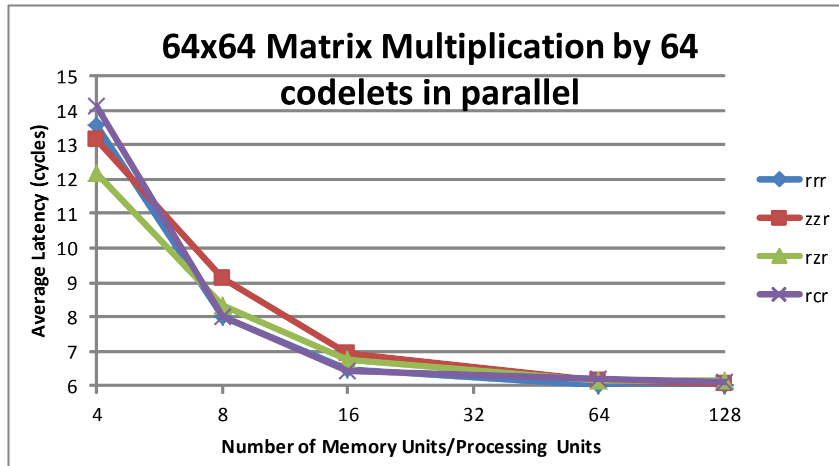


Figure 6.6: Memory network average latency vs number of memory unit in various layout

the number of cores. Moreover, each codelet only takes one of execution slots in the processing core. Given 64 parallel codelets executing on 16 cores with 4 slots on each, it only takes 7 cycles, 1 extra waiting cycle for each memory request. The figure reveals the suited ratio of the number of parallel codelets to the size of cores is between 4:1 and 8:1. Otherwise, the higher ratio causes excessive contentions on the network, and the lower ratio leads to underutilization. Consequently, Fresh Breeze system properly allocates 6 execution slots in one core.

After analyzing the architecture features, we compare the performance between different layouts under the Fresh Breeze architecture consisted of 16 processing cores integrating 4 execution slots in each, providing 64 codelet threads in total. All the cores equip with an auto buffer of 32 chunks and connect to the memory units through a network of 6 cycles latency.

The normalized runtime of the standard Matrix Multiplication is shown in the figure 6.7. The program in rcr layout(a row major matrix times a column major matrix providing results in row major), outperform baseline rrr layout 20 times. However, the

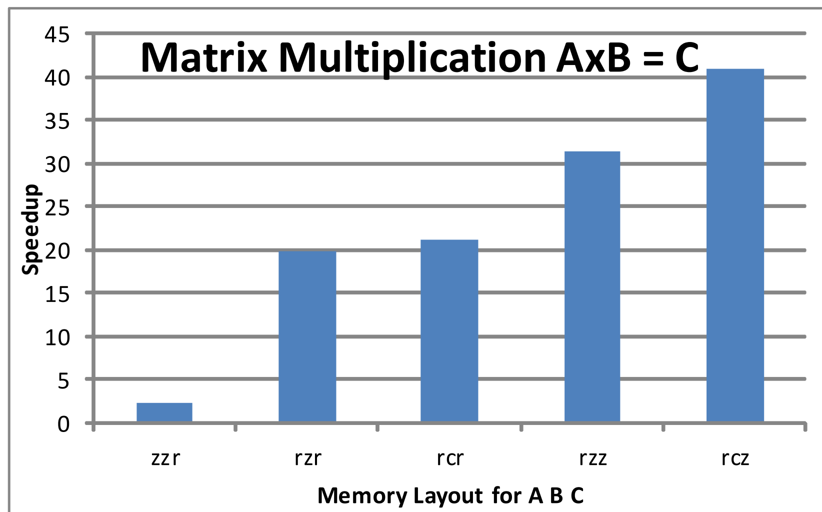


Figure 6.7: The performance on Matrix Multiplication, speedup vs baseline rrr layout

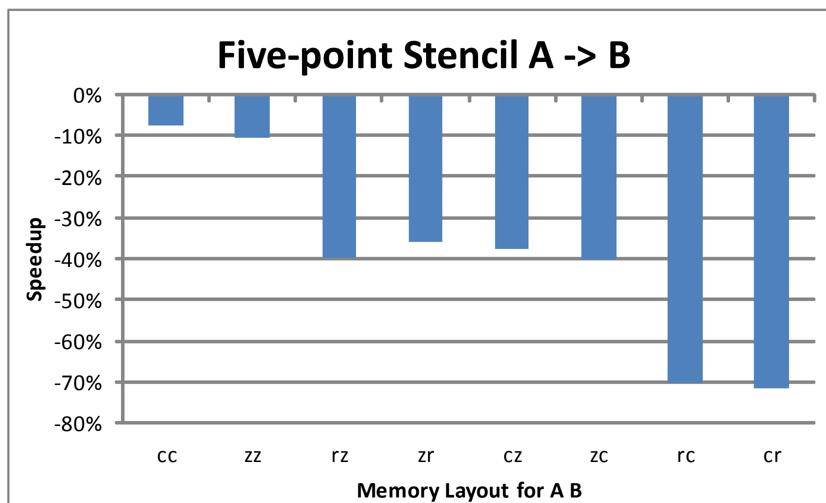


Figure 6.8: The performance on five-point stencil, speedup vs baseline rr layout

layout of the result matrix is another crucial factor for cache performance. The runtime of rzz layout is 37.3% faster than the one in rzr, and moreover the rcz layout is the optimal choice, which is 48% faster than the rcr and 40 times of the baseline rrr. The result matrix in the z morton improves the data locality by reuse both cached rows and columns. Actually, the program memory accessing behavior is similar to the block based matrix multiplication algorithm.

The other benchmark program five-point stencil consumes the data from adjacent elements, therefore, it highly benefits from the layout of good data locality. From figure 6.8, there is a severe performance penalty when the reading and writing directions are perpendicular to each other, such in the rc and cr layouts. The performance of cc and zz slightly degrade due to loops in the stencil code favored the row-by-row pattern. The compromised z morton is a eclectic layout adapted from the row and column major layouts. So, the performance is around half of the optimal result.

Chapter 7

CONCLUSION AND FUTURE WORK

In this project, we developed a templates based transform component in the compiler for MIT Fresh Breeze system. This transformation simplified codelet programming, bridges the conventional linear space memory model with Fresh Breeze's unique write-once and chunk based memory model. It compiles three classic linear algebra algorithms tested on the simulator and achieved good and sub liner scalable performance. It demonstrates that Fresh Breeze system has the ability of massive concurrent codelet execution and fine-grain resource sharing. We found the optimal configurations for major features guiding the processor design. The comparison between various layouts proves that Fresh Breeze system with codelet memory model takes advantages of the existing layout optimizations.

We plan to enhance the compiler transformation to support more generic program code and generate optimized memory layout. In this thesis, we only test a simple implementation of memory layout on the basic functions. However, in the real world applications, the data flow is formed by a chain of functions. In such programs, the output results is fed as other functions' input. The memory layout is not decided by the local optimal result from a single function, but by a group of functions. For the global optimizations, layouts commonly are chosen by heuristic approaches. In the future, firstly forming formula predicts the fresh breeze system performance directly from program code. secondly, study on how to choose the suitable combinations of layouts by the predicted metric. This approach will be the foundation for fresh breeze program optimization.

BIBLIOGRAPHY

[1] K. Arvind and Rishiyur S. Nikhil. Executing a program on the mit tagged-token dataflow architecture. *IEEE Trans. Comput.*, 39(3):300–318, March 1990.

[2] Siddhartha Chatterjee, Alvin R. Lebeck, Praveen K. Patnala, and Mithuna Thottethodi. Recursive array layouts and fast parallel matrix multiplication. In *Proceedings of the Eleventh Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '99, pages 222–231, New York, NY, USA, 1999. ACM.

[3] A. L. Davis and R. M. Keller. Data flow program graphs. *Computer*, 15(2):26–41, February 1982.

[4] Jack B. Dennis. Fresh breeze: A multiprocessor chip architecture guided by modular programming principles. *SIGARCH Comput. Archit. News*, 31(1):7–15, March 2003.

[5] Jack B. Dennis. Compiling fresh breeze codelets. In *Proceedings of Programming Models and Applications on Multicores and Manycores*, PMAM'14, pages 51:51–51:60, New York, NY, USA, 2007. ACM.

[6] Jack B. Dennis, Guang R. Gao, and Xiao X. Meng. Experiments with the fresh breeze tree-based memory model. *Computer Science - R&D*, 26(3-4):325–337, 2011.

[7] Jack B. Dennis, Guang R. Gao, Xiao X. Meng, Brian Lucas, and Joshua Slocum. The Fresh Breeze program execution model. In *Applications, Tools and Techniques on the Road to Exascale Computing, Proceedings of the conference ParCo 2011, 31 August - 3 September 2011, Ghent, Belgium*, pages 335–342, 2011.

[8] JackB. Dennis, GuangR. Gao, and XiaoX. Meng. Experiments with the fresh breeze tree-based memory model. *Computer Science - Research and Development*, 26(3-4):325–337, 2011.

[9] Jia Guo, Ganesh Bikshandi, Basilio B. Fraguela, Maria J. Garzaran, and David Padua. Programming with tiles. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '08, pages 111–122, New York, NY, USA, 2008. ACM.

[10] Joshua Suettlerlein, Stphane Zuckerman, and GuangR. Gao. An implementation of the codelet model. In Felix Wolf, Bernd Mohr, and Dieter an Mey, editors, *Euro-Par 2013 Parallel Processing*, volume 8097 of *Lecture Notes in Computer Science*, pages 633–644. Springer Berlin Heidelberg, 2013.

[11] Ying Ping Zhang, Taikyeong Jeong, Fei Chen, Haiping Wu, Ronny Nitzsche, and Guang R. Gao. A study of the on-chip interconnection network for the ibm cyclops64 multi-core architecture. In *Proceedings of the 20th International Conference on Parallel and Distributed Processing*, IPDPS'06, pages 64–64, Washington, DC, USA, 2006. IEEE Computer Society.

[12] Stéphane Zuckerman, Joshua Suetterlein, Rob Knauerhase, and Guang R. Gao. Using a "codelet" program execution model for exascale machines: Position paper. In *Proceedings of the 1st International Workshop on Adaptive Self-Tuning Computing Systems for the Exaflop Era*, EXADAPT '11, pages 64–69, New York, NY, USA, 2011. ACM.