Performance Analysis of Cache Oblivious Algorithms in the Fresh Breeze Memory Model

by

Joshua Foster Slocum

B.S., Massachusetts Institute of Technology (2010)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2011

© Massachusetts Institute of Technology 2011. All rights reserved.

Author

Department of Electrical Engineering and Computer Science
May 11, 2011

Certified by...

Jack B. Dennis
Professor Emeritus
Thesis Supervisor

Accepted by...

Dr. Christopher J. Terman
Chairman, Masters of Engineering Thesis Committee
Performance Analysis of Cache Oblivious Algorithms in the Fresh Breeze Memory Model

by

Joshua Foster Slocum

Submitted to the Department of Electrical Engineering and Computer Science on May 11, 2011, in partial fulfillment of the requirements for the degree of Master of Engineering in Computer Science

Abstract

The Fresh Breeze program execution model was designed for easy, reliable and massively scalable parallel performance. The model achieves these goals by combining a radical memory model with efficient fine-grain parallelism and managing both in hardware. This presents a unique opportunity for studying program execution in a system whose memory behavior is not well understood. In this thesis, I studied the behavior of cache-oblivious algorithms within the Fresh Breeze model by designing and implementing a cache-oblivious matrix multiply within the Fresh Breeze programming framework, as well as a cache-naive algorithm for comparison. The algorithms were implemented in C, using the Fresh Breeze run-time libraries, and profiled on a simulated Fresh Breeze processor. I profiled both programs across a range of problem sizes, memory speeds and memory types in order to best understand their behavior and accurately characterize their performance.

Thesis Supervisor: Jack B. Dennis
Title: Professor Emeritus
Acknowledgments

I would like to thank Professor Dennis for his help, guidance and patience. I would also like to thank Dr Xiaoxuan Meng and Mr Brian Lucas for their help with programming the FAST simulator. Finally, I would like to thank my family for their love and support.

This research was supported by NSF grant CCF-0937907.
# Contents

1 Introduction ................................................. 8
   1.1 Historical Context and Background .................. 8
   1.2 The Fresh Breeze Project .............................. 9
   1.3 Evaluating Matrix Multiply and Cache-Oblivious Algorithms .... 10

2 The Fresh Breeze Project .................................... 13
   2.1 The Fresh Breeze Program Execution Model .......... 13
      2.1.1 Chunks .......................................... 13
      2.1.2 Tasks ........................................... 15
   2.2 The Fresh Breeze Simulator System ................. 15

3 A Functional, Cache-oblivious Matrix Multiply ............ 17
   3.1 A Hierarchical Matrix Data Structure ............... 17
      3.1.1 Description ..................................... 17
      3.1.2 Asymptotic Analysis ............................ 18
   3.2 Matrix Addition ........................................ 19
      3.2.1 Algorithmic Description ......................... 19
      3.2.2 Asymptotic Analysis ............................ 21
   3.3 Recursive Matrix Multiplication ....................... 21
      3.3.1 Algorithm Description ............................ 21
      3.3.2 Asymptotic Analysis ............................ 23

4 Performance Comparison .................................... 24
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1 Data Collection Methods</td>
<td>24</td>
</tr>
<tr>
<td>4.2 Results</td>
<td>25</td>
</tr>
<tr>
<td>4.3 Discussion</td>
<td>28</td>
</tr>
<tr>
<td>4.4 Conclusion</td>
<td>30</td>
</tr>
</tbody>
</table>
List of Figures

2-1 Diagram of the control-flow for a typical spawn-join sequence. ........ 15

3-1 Each leaf node represents a 4x4 segment of a matrix ................. 18

3-2 A 6x5 matrix. ........................................................................ 19

3-3 Control flow diagram for the addition of two 8x8 matrices. Note how the parameter passed to spawn() determines where in the join chunk the child’s return value will be placed. ..................................... 20

4-1 The performance gain achieved by using a cache-oblivious algorithm, across varying L2 cache latencies. ................................. 27

4-2 The performance gain achieved by using a cache-oblivious algorithm, across varying main memory latencies. ................................. 27

4-3 Measured operations per second versus problem size for non-blocking memory with a latency of 20 system cycles. ......................... 28

4-4 The performance gain achieved by using a cache-oblivious algorithm, across varying main memory latencies. ......................... 29

4-5 The performance gain achieved by using a cache-oblivious algorithm, across varying main memory latencies. ......................... 29
List of Tables

2.1  Structure of a Chunk and its metadata. ........................................ 14

4.1  Timing accurate modeling of the row-based matrix multiply ............. 25
4.2  Timing accurate modeling of the cache-oblivious matrix multiply ....... 25
Chapter 1

Introduction

1.1 Historical Context and Background

Since the creation of the microprocessor in the early 1970's, until the last part of the 2000's, the serial performance of general-purpose processing units increased at an astonishing pace. A modern microprocessor is many orders of magnitude faster and more powerful than the early 4 or 8-bit systems of the 70's. Since the inception of the microcomputer, performance has been increased primarily through two methods: increasing the amount of computational work performed per clock cycle, i.e. improving pipelining, architecture or instruction sets; and decreasing the length of the clock cycle. By far, the greater contributor to increases in computational performance has been clock speed: while modern processors can pipeline approximately a dozen instructions at a time, clock speeds have improved several orders of magnitude over those utilized in the first generation of personal computers. Unfortunately, the physical limits of silicon-wafer based microprocessors place a soft-cap on how high chip frequencies can scale. Increases in frequency have caused the power consumption of microprocessors to grow exponentially, to the point where thermal energy dissipation has become the limiting factor in determining the maximum clock speed of such devices. [15]

In the past several years, major manufacturers of high-performance microprocessors have found themselves more and more constrained by the physical limits of silicon
transistor gates, and thus have had to search for alternative methods of increasing their processors' performance. Despite the limitations on clock speed, Moore's law has continued to hold: the number of transistors contained in a single processor has continued to grow at an exponential rate. The continued shrinking of microprocessors has allowed manufacturers to begin incorporating multiple execution cores on a single die of silicon. In an ideal scenario, this would result in perfect scaling of performance: multiplying the number of cores executing instructions could theoretically multiply the useful work performed by the same factor. In practice, achieving performance gains that even approach ideal scaling can be quite difficult. Worse yet, performance gains can vary widely across applications, because some computational tasks are inherently less parallelizable than others. The theoretical maximum performance gain is governed by Amdahl's law, which constrains the maximum achievable speedup from parallelization is inversely proportional to the non-parallelizable elements of a computation [16]. The sudden prevalence of multi-core processors in the high-performance computing market has spawned a renewed interest in software tools to help programmers exploit the abundance of additional processing cores on both commercial and consumer machines. Such tools range in complexity from rudimentary utilities that allow the user to directly manage the creation and scheduling of new threads, to sophisticated languages that abstract thread management away from the user completely. However, despite the advances in parallel performance, scalability and reliability made by these tools, modern parallel computing systems are still limited in scalability because they are still based on technologies originally designed for systems with serial execution.

1.2 The Fresh Breeze Project

The Fresh Breeze Project is an experiment which hopes to overcome the limitations of traditional computer systems by implementing a novel program execution model (PX M). A joint effort by three universities: MIT, Rice University and the University of Delaware, The Fresh Breeze Project aims to evaluate the potential benefits
of implementing hardware-level memory and thread management systems alongside powerful software tools for parallelism. The new memory architecture completely removes memory virtual management from the operating system, and enforces a write-once memory model. These modifications are expected to demonstrate improvements in multi-core scalability, reliability and performance. Fresh Breeze’s software tools includes cilk-style [12] spawn/join thread management, as well as compilers for higher-level functional languages such as FunJava [14], [3], [13]. It is hoped that these tools will increase the ease with which a programmer can incorporate multi-threaded execution into their programming [9]. At this point in the project, the effectiveness of the novel memory architecture is unproven and thus building an entire computing system is not practical. Instead, in order to assess the architecture’s potential benefits, a simulation of a Fresh Breeze system has been developed to run on the University of Delaware’s FAST; a simulator for the IBM Cyclops64 processor [4]. MIT, Rice University and The University of Delaware are all currently involved in the development of a software platform that will allow for the writing and execution of application-level code on the simulated architecture. This platform includes compilers for several programming languages, a software kernel that handles interfacing with the hardware, and a run-time library that provides user access to the storage system and manages the scheduling of threads. At the time of this thesis’ inception, development had proceeded to the point where user-level code could be compiled and run on the simulated system, allowing for timing-accurate measurements of the system’s performance.

1.3 Evaluating Matrix Multiply and Cache-Oblivious Algorithms

There are two primary contributions to this thesis. The first is to provide a Fresh Breeze implementation of a matrix multiply program that can be used to profile the system’s performance. Data from such profiling constitutes evidence for evaluating
the potential of the Fresh Breeze model, building on previous efforts [9]. Additionally, the implementation would provide a model on which further matrix manipulation programs could be based. The second major contribution of this thesis is to evaluate the potential of cache-oblivious algorithms within the Fresh Breeze program execution model. Cache oblivious algorithms achieve asymptotically optimal cache-behavior without the need for tuning the implementation for specific memory configurations [11]. It has been proven that cache-oblivious algorithms implemented with Cilk-style parallelism enjoy a superior asymptotic bound on cache performance versus a 'cache-naive' implementation [10], but an asymptotically lower bound does not always translate into superior performance for realistically-sized problems. The cache-oblivious matrix multiply implementation allows the merits of such algorithms to be experimentally tested on the Fresh Breeze system; the implementation also serves as a model on which other Fresh Breeze style cache-oblivious algorithms could be based. These contributions will enhance understanding of computation within the Fresh Breeze program execution model, provide data about the potential performance of a system implementing that model, and provide a foundation upon which further similar work can be performed.

Fresh Breeze's write-once memory model enforces a purely functional style of programming. There is little literature on the implementation or performance of functional cache-oblivious algorithms: the algorithmic design and profiling performed in this thesis thus constitute novel work, and correspondingly introduce several novel challenges to be overcome. Many of these challenges arise from the write-once memory model; virtual memory locations are immutable, and thus for computation to proceed, methods must return new values instead of relying on mutation. Furthermore, the memory model constrains all data structures to take the form of directed, acyclic graphs [8]. Implementing a Fresh Breeze style cache-oblivious matrix multiply thus requires both heavy modification to the traditional 'divide-and-conquer' implementation of cache-oblivious matrix multiply, as well as the design of a new data structure that stores a matrix as a DAG in a manner that facilitates both efficient division of the problem among threads and optimal memory access patterns. The
cache-naive matrix multiply should be implemented with a data-structure designed to suit the cache-naive algorithm’s memory access patterns, allowing for a more ‘fair’ performance comparison.

Programming for the simulated Fresh Breeze system presents its own set of complications: because the platform is still being actively developed, many common programming tools are not yet available; the simulator had no built in support for debugging or performance profiling. In addition to slowing the development cycle, this dearth of tools places limits on the depth of obtainable data. The Fresh Breeze simulator system provides limited support for instrumentation; the data to be obtained will take the form of measurements of program running time and histograms of processor utilization. These values are measured across a range of parameters, including number of available cores, problem size to be solved, memory speed, and memory type. These values will demonstrate the system’s scalability and suitability for multiple levels of memory.

The success of this thesis should be judged by its contribution to the understanding of the Fresh Breeze program execution model’s run-time behavior, the efficiency of the designed algorithms and data structures, and the absolute performance achieved by the matrix multiply implementations. The body of the thesis is organized into three sections: an in-depth description of the Fresh Breeze program execution model, a design description of the algorithm and data structure used to implement a cache-oblivious Fresh Breeze style matrix multiply, and a detailed presentation and discussion of the performance tests’ results. The accompanying appendices present the ‘naive’ matrix multiply algorithm used in the performance comparison, and a more formal asymptotic analysis of the data-structures and algorithms used in implementation. The simulation results of this research have been previously reported in two joint papers, authored by Jack Dennis, Xiao Meng, Guang Gao, Brian Lucas and myself [6] [5].
Chapter 2

The Fresh Breeze Project

2.1 The Fresh Breeze Program Execution Model

The Fresh Breeze Model for computation encompasses both an application programming interface for expressing concurrency and a novel memory system model. It is designed to be specifically suited for massively parallel computing systems. The memory model constitutes a radical departure from traditional computing systems at the hardware level, and the hardware directly supports a simple but powerful method for expressing parallelism.

2.1.1 Chunks

The Fresh Breeze memory model enforces a write-once policy for any data that an execution unit wishes to transfer out of its cache. To implement this, the memory model enforces the discipline that all data above the CPU execution level exists in the form of chunks. A chunk is a fixed-size unit of memory. All chunks are the same set length (in the range of 128-1024 bytes; the current implementation uses 128), and each chunk has a unique, global 64-bit identifier, known as the chunk's handle. A bit map is kept as part of the system state, identifying which handles are currently in use. Initially, a chunk may be read or written to only by the task that created it. Once the task has finished writing to the chunk, it may either be deallocated, or
A sealed chunk may be read by any task using the chunk’s handle, but no task may write to it. Thus, on a low level, Fresh Breeze enforces a purely functional model of data. The collection of all chunks and pointers in memory forms a directed graph, with pointers as edges and chunks as nodes. Because a pointer may only be written into a chunk when it is first created, the graph is necessarily acyclic, and thus garbage collection may be performed simply using reference counts [7].

If it were defined as a Java class, a chunk would appear as the code shown below in table 2.1. The values array contains 16 64-bit values, each of which may be used to store either a handle, a 64-bit value, or two 32-bit values. The tags array contains sets of boolean tags, with each set corresponding to an element in the values array. These tags indicate whether the element contains valid 32 or 64-bit data, or a handle.

The type field specifies whether the chunk contains data; executable code, or data structures used for program execution. A chunk’s ref_count is incremented when it is first written, and every time its handle is incorporated into another chunk; the count is decremented when either the creating task has finished accessing the chunk, or when another chunk containing this chunk’s handle is collected. Finally, the ref_count field includes how many other chunks contain references to this chunk; it is used primarily for garbage collection purposes. A running task may create a new chunk by acquiring a new, unique handle, and then calling the chunk_write() instruction on the handle and the data to be stored. Once a chunk has been written, any thread with knowledge of the chunk’s handle can access the chunk by calling the read() instruction on its handle [7].
2.1.2 Tasks

The Fresh Breeze system supports very fine-grain parallelism, implemented similarly to the Cilk [12] interface for parallelism. A task is a small unit of computation that may be executed by a processor core. Any task may spawn more tasks to further divide computational work. If a task wishes to spawn child tasks, it must first instantiate a join-ticket, which is a special kind of chunk in which child tasks may report their return values. The call to instantiate a join ticket has three arguments: the number of children to be spawned, a continuation function which will be executed upon completion of the children, and arguments to the continuation function. The call for spawning a child task has three arguments: a pointer to the function the task will execute, a struct containing the function’s arguments, and an integer specifying where in the join-ticket the child should write its return value. When a child task completes its execution, it calls join-update, which writes a value into one of the elements of its the join-ticket. When all children have completed execution, the continuation function is executed as a new task with access to the information contained in the join-ticket. A control flow diagram of this spawn-join-continue process is shown in figure 2-1. In a full implementation of a Fresh Breeze computer system, the primitive functions for manipulating tasks would be implemented as hardware instructions.

2.2 The Fresh Breeze Simulator System

The Fresh Breeze system is simulated on top of the FAST [4] simulation of an IBM Cyclops 64 many-core processor. Aside from on-chip networking, the FAST simulator
is cycle-accurate. The Cyclops 64 processor consists of eighty pairs of thread-units, networked with each other via a crossbar switch, which can also provides several access ports to external DRAM. Thus there are 160 thread units in total. There is an instruction cache that holds executable code associated with each group of five thread unit pairs. Each thread unit has its own 30 KB block of SRAM, and shares a floating point unit with its partnered thread unit. In order to execute the Fresh Breeze memory model, many of the thread units are dedicated to simulating the storage system. In addition to the aforementioned thread and storage interfaces, the run-time libraries also provide support for standard Cyclops mathematical operations and processor-related utilities.

Simulation of a Fresh Breeze-style system is achieved in software by assigning each Cyclops thread unit to simulating different elements of the thread and memory systems. A group of forty thread units is used to model a group of forty Fresh Breeze thread units and their L1 caches. Across the cross-bar switch, the majority of thread units are dedicated to emulating a second level of memory storage. These main storage units (MSU's) may be set to represent either an L2 cache or main memory by changing whether or not tasks block when accessing it. The scheduling unit is used to maintain a global work-stealing table from which idle EU's may acquire more tasks, and the profiling unit maintains a record of pertinent information about the behavior of other units during simulation.
Chapter 3

A Functional, Cache-oblivious Matrix Multiply

3.1 A Hierarchical Matrix Data Structure

3.1.1 Description

As previously mentioned, the Fresh Breeze memory model enforces that each data structure be in the form of a DAG, with chunks as nodes and handles as edges. Since each chunk is 128 bytes, and a handle is 8 bytes, each chunk may point to no more than 16 other chunks. There are many possible ways a matrix could be represented within such constraints. The data structure presented in this section was specifically designed for a divide-and-conquer approach to matrix operations. Matrices are represented by a hierarchical tree structure, with the elements in each level of the tree encompassing larger and larger parts of the matrix as one moves up the tree. The root node of the tree represents the entire matrix.

We begin by examining the structure of leaf nodes, which hold the actual data associated with a matrix. The chunk of a leaf node contains an array of 16 64-bit data values. This is treated as a 4x4 array in row-major format, as illustrated by Figure 3-1. A leaf node represents a fixed-size window that encompasses a small region of a matrix. Thus a matrix of size 3x4 could fit entirely into a single leaf node, but not
one of size 5x5 or 2x7.

Instead of containing data, parent nodes contain handles that point to other nodes. The internal structure of a parent node is similar to a leaf node, except that each element represents a smaller submatrix instead of a data element, thus making the size of the parent nodes' "window" proportional to its height in the tree. Children of a parent node are arranged for spatial locality: the "window" of element '1' of a parent node is contiguous with the windows of elements '2' and '5'. This setup is similar to a quadtree[1], except that each parent has sixteen children instead of four. Matrices are constructed so that the top-left portion of the tree (corresponding to children with smaller row and column coordinates) is filled in first. Figure 3-2 shows how a 6x5 matrix would be represented by this data structure.

3.1.2 Asymptotic Analysis

This section will provide a brief analysis of the asymptotic storage usage of this data structure as well as the work required for a few relevant tasks. The upper bound on the total amount of storage used by a matrix under this scheme can be found by examining the worst case scenario: storing a vector. This is the 'worst-case' scenario because only one fourth of the total storage of each chunk in the tree will be utilized. For a vector of size $n$, $n/4$ leaf-level chunks will be required to store the vector, and $n/16$ chunks will be required to store those chunks; the $k$th tree level will be of size $n/4^k$, thus giving the recurrence: $T(n) = T(n/4) + \theta(n)$. Therefore, by the master method[2], the total storage used will be $\theta(n)$.

Random access of a data element requires traversing the tree, starting at the root node. The total work involved is thus equal to the height of the tree. Since each level of the tree encompasses at least 4 times more elements than the next level down, the total height of a tree for a matrix with $n$ as its largest dimension is $O(log_4(n)) = O(log(n))$. Thus the random access time for an individual element is
\[ \begin{bmatrix} 1 & 2 & 3 & 4 & 1 \\ 5 & 6 & 7 & 8 & 5 \\ 9 & 10 & 11 & 12 & 9 \\ 13 & 14 & 15 & 16 & 13 \\ 1 & 2 & 3 & 4 & 1 \\ 5 & 6 & 7 & 8 & 5 \end{bmatrix} \]

**Figure 3-2: A 6x5 matrix.**

\(O(\log(n))\). Traversing the entire tree takes an amount of work equal to the number of elements in the tree: \(\theta(n)\). Creating another tree of equal size will therefore also take \(\theta(n)\) work.

### 3.2 Matrix Addition

#### 3.2.1 Algorithmic Description

The implementation of matrix multiply presented in this paper requires the addition of matrices as a step. This section presents an algorithm for adding matrices stored in the data structure discussed above. The presentation of a matrix addition algorithm
serves both to document the specific implementation used and as an example on which the explanation of multiplication may build. To multiply two or more matrices, we start at the root node of each matrix. We then split the matrices into smaller parts, assigning a new task to handle the multiplication of each pair of parts, and assign a continuation function that assembles the sums back into a larger matrix. Each task continues to recursively split the matrices, spawning more children for each submatrix, until the leaf nodes are reached. A leaf task sums its two leaf chunks, and allocates a new chunk in which the results are stored. The handle for this sum chunk is written into the join ticket in the element corresponding to the leaf chunks’ position in their parents’ chunk. The parent’s continuation thus needs only to call join-update on its join chunk, sending the result to its parent’s continuation; the process continues all the way back up the recursion tree until the original tasks’s continuation has access to a chunk which points to the sum matrix. In this way the structure of the sum matrix is copied from the other matrices, but the sum matrix’s data is the sum of the others’. Figure 3-3 demonstrates how one step down and up the recursion tree might look for the sum of two matrices.

![Control flow diagram for the addition of two 8x8 matrices. Note how the parameter passed to spawn() determines where in the join chunk the child’s return value will be placed.](image)

Figure 3-3: Control flow diagram for the addition of two 8x8 matrices. Note how the parameter passed to spawn() determines where in the join chunk the child’s return value will be placed.
3.2.2 Asymptotic Analysis

This algorithm traverses the tree of each matrix, performs a constant-time operation for each element in each matrix, and then creates a new matrix of the same size. The total work for $m$ matrices with largest dimension $n$ is therefore $\theta(n) + \theta(m \cdot n) + \theta(n) = \theta(m \cdot n)$. Because no intermediate data structures are created, the space utilization is $\theta(n)$. The span for a parallel implementation is simply the height of the tree times the number of matrices: $\theta(m \cdot \log(n))$, giving parallelism $\theta(n/\log(n))$.

3.3 Recursive Matrix Multiplication

3.3.1 Algorithm Description

This section will begin by examining the a simple case: multiplying two matrices of size 4x4. Matrices of this size are small enough to fit into a single chunk, and thus do not need to be stored as a tree. Let it be that for matrices $A$, $B$ and $C$ of size 4x4, $A \cdot B = C$. We thus have that each element of $C$, $C_{i,j}$, is the dot-product of row $i$ of $A$, $A_{ri}$, and a column $j$ of $B$, $B_{cj}$:

$$C_{i,j} = A_{ri} \cdot B_{cj}.$$  

Calculating $C$ is thus simply a matter of computing the dot product of each row of $A$ with each column of $B$. This procedure will not work on larger matrices because their data structures are multi-level. We will define a new operator, matrix-dot, represented by the $\square$ symbol, which will allow us to perform a recursive computation similar in form to the leaf case. For two arrays of matrices, $V_1$ and $V_2$, where $V_i^1$ and $V_i^2$ are matrices of size $n \times n$, let $V_1 \square V^2$ be defined as:

$$V^1 \square V^2 = \sum_i V_i^1 \cdot V_i^2$$

Which will return a matrix of size $n \times n$. Using this operation, we can define a recursive procedure for multiplying matrices stored in hierarchical tree structures. Let
X and Y be the root nodes of two n x n matrices, where n is some power of 4 greater than 1. Each element of X and Y is thus a handle pointing to a smaller submatrix of size n/4 x n/4. We can then define a procedure, matMultCO, which multiplies X and Y by computing the matrix-dot of each row and column of X and Y. The □ of each row, column pair can be calculated by recursively applying matMultCO to each pair of submatrices and summing the results. For X * Y = Z, we thus have:

\[ Z_{i,j} = X_{ri} \square Y_{cj} = \sum_{k=0}^{3} X_{ri[k]} \ast Y_{cj[k]} \]

In pseudocode, this might look like:

```pseudocode
matMultCO(X, Y, size) {
    for(row =0; row < 4; row++) {
        for(col =0; col < 4; col++) {
            if(size <=4) {
                //for leaf-nodes, perform scalar dot-product
                Z[row,col] = dot(X,Y,row,col)
            } else {
                //for parent-nodes, perform matrix dot-product
                Z[row,col] = matrix-dot(X, Y, row, col, size)
            }
        }
    }
    return Z;
}

matrix-dot(X, Y, row, col, size) {
    products = matrix-array[4];
    for(elt = 0; elt < 4; elt++)
        products[elt] = matMultCO(X[row,elt], Y[elt,col], size/4)
    return sum(products);
}
```
Multiplying Arbitrarily Sized Matrices

Generalizing to multiplying matrices of arbitrary sizes incurs some complications that must be noted. First, one needs to keep careful track of exactly how large each matrix and its submatrices are, lest one attempt to access a null pointer. Doing so requires trivial arithmetic, but it must be carefully implemented. More interesting is the case where the two matrices X and Y are represented by trees of different height. For example, if X is of size 80 x 8 and Y is of size 8 x 4, then X will have a tree height of 4 where Y will have a tree height of only 2. In this particular case, matrix multiply must be called recursively on each of the submatrices of X and the whole of Y. This recursion continues until the two multiplicands have trees of the same level, and then multiplication proceeds as described in the previous section. In the case where the multiplicands’ tree height is greater than the product’s, matrix-dot is called recursively on the multiplicands until they are at the same tree height as the product matrix.

3.3.2  Asymptotic Analysis

For multiplying two \( n \times n \) matrices, the total amount of work is equal to the cost of 16 calls to dot-matrix plus the cost of assembling the sums of those products. This work is described by the recurrence: \( T(n) = 64 \times T(n/4) + \theta(n) \). The master method thus gives total work \( \theta(n^3) \), the same as a normal, naive matrix multiply. The span is given by the total work done on one path of the recursion tree, giving \( S(n) = S(n/4) + \theta(n) = \theta(n) \) by the master method. The parallelism is therefore \( \theta(n^2) \). Because each multiplication must store its result in a newly created matrix object, the space utilization is described by the same recurrence as the amount of work, giving a total usage of \( \theta(n^3) \) for all intermediate data structures.
Chapter 4

Performance Comparison

4.1 Data Collection Methods

Using the Fresh-Breeze C runtime libraries, I implemented two versions of matrix multiply: a cache oblivious version using the algorithm described in chapter 3, and one implementing a recursive version of naive matrix multiply. Both programs were fully parallelized and stored all data in the chunk-based memory system. Because the Fresh Breeze Simulator is implemented on the software level, it was necessary to "pad" critical functions with idle cycles, as done in [9], to achieve timing-accurate simulation. All data was taken from a Fresh Breeze simulation implementing 40 thread units.

Padding allows the simulation to be run with the desired timing ratios between various actions, as summarized in tables 4.1 and 4.2. Dividing the runtime measured in simulated cyclops cycles by the ratio between simulator cycles and Fresh Breeze system cycles will thus yeild an approximation of the total number of Fresh Breeze system cycles taken by each computation. Data was collected for two forms of simulated memory: blocking memory, which causes tasks to be suspended upon a chunk_read, simulates DRAM level memory; non-blocking memory requires tasks to wait idly until the requested chunk is fetched, simulating an L2 on-chip cache. To measure the total running time of an execution, the program simply prints the system clock value at the beginning and end of computation. This gives the running time in
Table 4.1: Timing accurate modeling of the row-based matrix multiply

<table>
<thead>
<tr>
<th>action</th>
<th>system cycles</th>
<th>simulation cycles</th>
<th>padding</th>
<th>total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Task Startup</td>
<td>4</td>
<td>262</td>
<td>378</td>
<td>640</td>
</tr>
<tr>
<td>Leaf Task Compute</td>
<td>32</td>
<td>376</td>
<td>4744</td>
<td>5120</td>
</tr>
<tr>
<td>Task Switch Save/Restore</td>
<td>16</td>
<td>262</td>
<td>2298</td>
<td>2560</td>
</tr>
<tr>
<td>L1 Cache Access</td>
<td>32</td>
<td>3047</td>
<td>2073</td>
<td>5120</td>
</tr>
</tbody>
</table>

Table 4.2: Timing accurate modeling of the cache-oblivious matrix multiply

<table>
<thead>
<tr>
<th>action</th>
<th>system cycles</th>
<th>simulation cycles</th>
<th>padding</th>
<th>total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Task Startup</td>
<td>4</td>
<td>262</td>
<td>138</td>
<td>400</td>
</tr>
<tr>
<td>Multiply Leaf Task</td>
<td>152</td>
<td>1972</td>
<td>13228</td>
<td>15200</td>
</tr>
<tr>
<td>Add Leaf Task</td>
<td>104</td>
<td>1772</td>
<td>8628</td>
<td>10400</td>
</tr>
<tr>
<td>Task Switch Save/Restore</td>
<td>16</td>
<td>262</td>
<td>1338</td>
<td>1600</td>
</tr>
<tr>
<td>L1 Cache Access</td>
<td>32</td>
<td>3047</td>
<td>153</td>
<td>3200</td>
</tr>
</tbody>
</table>

simulated Cyclops 64 cycles.

Tables 4.1 and 4.2 demonstrate how the padding scheme works: each action should take a known number of cycles in the Fresh Breeze system. By measuring how long each action takes in the simulator, one can determine the amount of padding required to make the ratio of the actions' runtime ratios equal to those predicted for an actual Fresh Breeze system. When the total number of simulation cycles for a program is measured, finding an estimate for the number of Fresh Breeze cycles is then merely a matter of multiplying this value by the ratio of simulator to system cycles for the given program. In the case of the row-based matrix multiply, this value is .00625; for the cache-oblivious version, .01. The values in table 4.1 are identical to those found in table 1 of Dennis et al [9]; this is because the row-based matrix multiply makes use of the dot-product program for multiplying a row by a column. To find the required padding given a desired latency in Fresh Breeze system cycles, one need merely multiply by the cycle ratio and subtract the number of Cyclops simulation cycles taken by the simulator routine.

4.2 Results

For both types of memory, each matrix-multiply implementation was profiled across a range of values for problem size and memory speed as measured in Fresh Breeze.
system cycles. Figures 4-1 and 4-2 show the relative speedup of the cache-oblivious version versus the naive version for each type of memory. Both figures demonstrate a significant performance gain from using the cache-oblivious implementation. It is important to note, though, that the speedup quickly drops as problem size increases. Looking at figure 4-3, it can be seen that this is due not to the cache-oblivious algorithm becoming slower, but due to the cache-naive algorithm becoming faster. Unfortunately, multiplying two 48x48 matrices approaches the current Fresh Breeze simulator’s limits; attempting larger inputs results in either a crash or an indefinite hang due to memory constraints. A more complete examination of the scalability of the two programs will thus require a simulator with greater memory capacity. However, it is still readily apparent that for smaller problem sizes, the performance of cache-oblivious matrix multiply significantly exceeds the cache-naive version. Of some concern is the cache-oblivious algorithm’s $\Theta(n^3)$ memory usage for storing intermediate values. It is plausible that such usage could make the use of cache-oblivious algorithms impractical or simply slow for large problem sizes. In such a case, the optimal solution for large matrix multiplications would be to adopt a hybrid approach, where multiplication proceeds using the cache-naive algorithm for higher levels of the storage tree, then switches to the cache-oblivious version below a certain tree-height threshold.

Figures 4-4 and 4-5 show measured processor utilization levels for the cache-oblivious matrix multiply. These figures demonstrate that despite having good overall performance, the cache-oblivious matrix multiply is still bottlenecked by memory. Fresh Breeze’s task switching is fast enough and the task granularity is low enough that the overhead of task-switching alone can not explain the low utilization rates. Despite this low processor utilization, the program still achieves very respectable performance. Multiplying two 48x48 matrices with an L2 cache latency of 20 system cycles takes approximately 30,000 system cycles on a simulated 40-core Fresh Breeze chip. This multiplication requires $48^3$ adds and multiplies, or 221,184 operations in total. Assuming a 500MHz clock rate [9], this equates to a raw performance of approximately 3.7 GOPs. For the simulated execution units, the theoretical maximum
Figure 4-1: The performance gain achieved by using a cache-oblivious algorithm, across varying L2 cache latencies.

Figure 4-2: The performance gain achieved by using a cache-oblivious algorithm, across varying main memory latencies.
performance of a leaf-level matrix multiply is determined by the 32 cycles required to read elements from the L1 cache, and the 128 cycles required to perform 64 pipelined multiplies and 64 pipelined additions. This gives \((128 \times 500)/160 = 400\) MOPs per execution unit, or 16 GOPs for the simulated system. The system is thus able to achieve approximately 25% of theoretical maximum performance with a memory system managed entirely by hardware.

4.3 Discussion

Although the cache-oblivious matrix multiply achieves impressive performance for the problem sizes presented here, there are legitimate concerns about its scalability, due to memory usage. Multiplying two 1024x1024 matrices would require gigabytes of storage for intermediate values, where the matrices themselves only take a few megabytes. Theoretically, the storage requirements should be manageable: the algorithm’s parallelism is \(\theta(n^2)\) and so only \(\theta(n^2)\) memory should be in use at any point.
Figure 4-4: The performance gain achieved by using a cache-oblivious algorithm, across varying main memory latencies.

Figure 4-5: The performance gain achieved by using a cache-oblivious algorithm, across varying main memory latencies.
in time. Large matrix multiplies will thus test the system’s ability to handle large amounts of transient data. Manipulation of such large matrices is quite common in applications such as image compression and computer vision, and so performance at that scale is a very important performance metric. In order to prove the Fresh Breeze program execution model’s viability as a scalable platform for massive parallelism, future simulators will need to demonstrate Fresh Breeze systems capable of handling such problems efficiently. The performance demonstrated in this paper is similar to that found in [9], reinforcing the conclusion that the Fresh Breeze PXM is worthy of further study and development.

4.4 Conclusion

The Fresh Breeze program execution model shows promise as a platform for massively parallel computing. This thesis strengthened that promise by demonstrating that the model is capable of supporting sophisticated algorithm design and that through such design one can achieve good parallel performance. In particular, the divide-and-conquer paradigm for cache-oblivious algorithms was shown to be directly applicable to the Fresh Breeze memory model, resulting in a performance speedup of several factors. There are many avenues down which future work might travel. An obvious route is to expand the simulator system, increasing the memory to allow for problems of greater size to be profiled. Another promising project would be to expand the simulator’s instrumentation, thus allowing more sophisticated study of the memory system. Further algorithm development is also warranted; there may be other cache-oblivious implementations that do not incur such a high memory cost. This thesis is part of a growing body of work demonstrating the that the Fresh Breeze system holds great opportunity for study.
Bibliography


