ASYNCHRONOUS MATRIX FRAMEWORK WITH PRIORITY-BASED PROCESSING

A Thesis

Presented to

the Faculty of

California Polytechnic State University, San Luis Obispo

In Partial Fulfillment

of the Requirements for the Degree

Master of Science in Computer Science

By

Jeremy Andrew Seeba

May 2007

AUTHORIZATION FOR REPRODUCTION
OF MASTER'S THESIS

I grant permission for the reproduction of this thesis in its entirety or any of its parts, without further authorization from me.

_____
Signature

_____
Date

ABSTRACT

Asynchronous Matrix Framework with Priority-Based Processing

Jeremy Andrew Seeba

Complex computer graphics simulations rely on a large number of calculations which can usually be represented through matrix operations. In a real-time simulation, not all elements in a matrix calculation would need to be known synchronously as some elements may be off screen at the time. This thesis investigates asynchronous computation of matrix results and the impacts of the framework compared to a synchronous design.

For the purpose of asynchronous execution, matrix operations can be seen as base operators, such as multiply or add, combined with the operand matrices on either side of the operator. Instead of performing the operation when it is first seen, the operator and operands are stored, space is allocated for the results, and tasks necessary to complete execution of the resulting matrix can be made. By setting up operations this way, execution of operations are completely decoupled from the initial operation and different execution strategies can be employed.

This paper looks at different strategies such as a just-in-time execution where results are only computed when a matrix element is requested and multi-threaded execution using both priority and non-priority based execution. Each thread can implement actual execution in different ways such as on the CPU or on a GPU. The flexibility of asynchronous matrices is weighed against the overhead needed to make this system function.

Table of Contents

Tables

Table

Page

Figures

Figure

Page

CHAPTER 1

Introduction

Complex physical simulations rely on massive computations to complete each simulation step. For a large number of simulations, the interactions between various simulation elements and forces that link them together can be represented by matrices and matrix operations, such as the visualization of smoke [8] or cloth simulation [4]. The number of matrix elements scales with the geometric complexity of objects causing significant slow down for large simulations. Real-time interaction is a desirable property of many physical simulations (e.g. for computer games) and large matrix computations are a challenge to keeping real-time interaction possible. This thesis investigates a possible solution by allowing for only some of the matrix elements to be computed for a given frame. We call this an *asynchronous* framework. This framework is justified by the fact that real-time simulations may only need to compute the interactions for elements that would be displayed on the screen for a given time frame, while other elements of the model may be off screen for that frame. This means that the viewing space versus and the simulation space is vastly different, with the viewing space likely being much smaller. When working in a constrained viewing space where only a few elements of the entire original simulation are used, there could be a number of elements in the simulation that are not needed. Instead of rewriting a simulation to optimize for only the pertinent elements, the matrix elements that are not needed can be delayed until after useful data is collected and possibly never calculated if the simulation never displays those elements. Likewise, this type of asynchronous framework allows for a de-coupling of rendering and

computing the next frame. In other words, for a given frame, the geometric/matrix elements on screen can be computed and, while those elements are drawn, the rest of the matrix elements can be updated. In traditional matrix frameworks this is not possible but with an asynchronous matrix framework it is possible.

Before we discuss the project in more detail, please note that throughout this paper upper-case, bold letters will be used to represent matrices. A traditional matrix framework will be used to mean a synchronous, non-multithreaded framework. A synchronous matrix calculates all of its elements when an operation is defined, as in **A*****B**. An asynchronous matrix calculates its elements at *some point* after the operation is defined. So if **C**=**A*****B** and **C** is an asynchronous matrix, **C** would store its operands as **A** and **B** and store information they need to be multiplied together at a later time to complete the elements of **C**. For most of this paper the example **C**=**A*****B**, **E**=**C**+**D** will be used where **A**, **B**, and **D** are assumed to have numerical data loaded directly into them.

While using synchronous matrices there are many different issues that are not desirable. The first and possibly most obvious is that a matrix operation calculates all the elements of the resulting matrix at the time the operation is called. The main reason this is undesirable is because, if the operation is called in the main thread of execution, there is no other work that can be done until *after* the computation is finished. This can be seen in the following pseudo-code:

```
C=A*B
Get User Input    **completely unrelated to C**
Update Screen     **completely unrelated to C**
```

Even through getting user input and updating the screen are unrelated to the matrix operation, both have to wait until the operation completes before the program would

become responsive again to a user. Another issue with synchronous execution is that no individual elements are ready to be used until the *entire* matrix is ready. In other words, even if element $\mathbf{C}_{0,0}$ is useful to determine what to do next in a program, it is not available until *all* other matrix elements are also computed.

Even though there are some drawbacks to synchronous matrix execution, there are some benefits which simplify bookkeeping and associated overhead that asynchronous matrices incur. For example, a synchronous matrix does not have to track if a previous matrix is finished before executing on the data in that matrix because synchronous matrices must have all of its elements finished by the time it is available for use in other matrix operations. Because of this, more complex algorithms can be applied without checking whether certain sections of a matrix are finished. Another benefit is that the amount of space that is necessary in a synchronous matrix is exactly the size of the number of elements the matrix contains and does not need any additional bookkeeping information like asynchronous matrices need. Complexity related to synchronization is not present in a traditional matrix framework, though a multi-threaded synchronous matrix framework would still need to some extra mechanisms to perform synchronization such as locks.

Even with the reduced complexity and other benefits of a synchronous matrix framework, we found that the overhead required for asynchronous matrices offers many benefits. Specifically, our algorithm provides the ability to access 50% of a matrix's elements between 41% and 79% faster than a synchronized matrix framework. We discuss these results and more later in section 4. One of the major requirements of the asynchronous matrix framework we present is that it must represent matrices in a fashion

that is as similar as possible to how a synchronous matrix framework would while only adding the necessary interfaces to account for difference in execution styles. Another major requirement is to completely decouple the definition of a matrix via a matrix operation, such as **C**=**A***__B__**,** from the execution of any of the elements and decouple the execution of elements from one another such that as an example $\mathbf{C}_{0,0}$ might not be completed but $\mathbf{C}_{3,4}$ could be complete and there is no interdependence.

Our system is able to provide fast asynchronous matrix computation through the use of multi-threading to allow for background processing and utilization of unused resources. Because of this, some additional data structures and computation, such as prioritizing what computation to do next, need to be added to our matrix framework representation to help facilitate multithreading. In summary, the overall goal of this project is to create a matrix framework that is:

- comparable in speed to a synchronous matrix
- quicker to give individual elements
- able to out perform comparable traditional matrix frameworks using multi-threading.

Comparable speed to traditional matrix frameworks is achieved through limiting the overhead that is associated with each element of the matrix compared to the total computation required for that element. Individual elements are retrieved from an asynchronous matrix much more quickly because computation is not dependent on any other matrix elements and the matrix does not have to complete all the elements before returning any elements like a synchronous matrix has to. Unless the overhead is too great compared to the computation of matrix elements, multi-threading allows asynchronous

matrices to out perform traditional matrix frameworks because more computational resources are available. Multi-threading allows for multiple jobs, which represent parts of the matrix that need to be computed, to be executed at the same time as long as they do not depend on any other jobs completing before they can execute.

1.1 Architecture

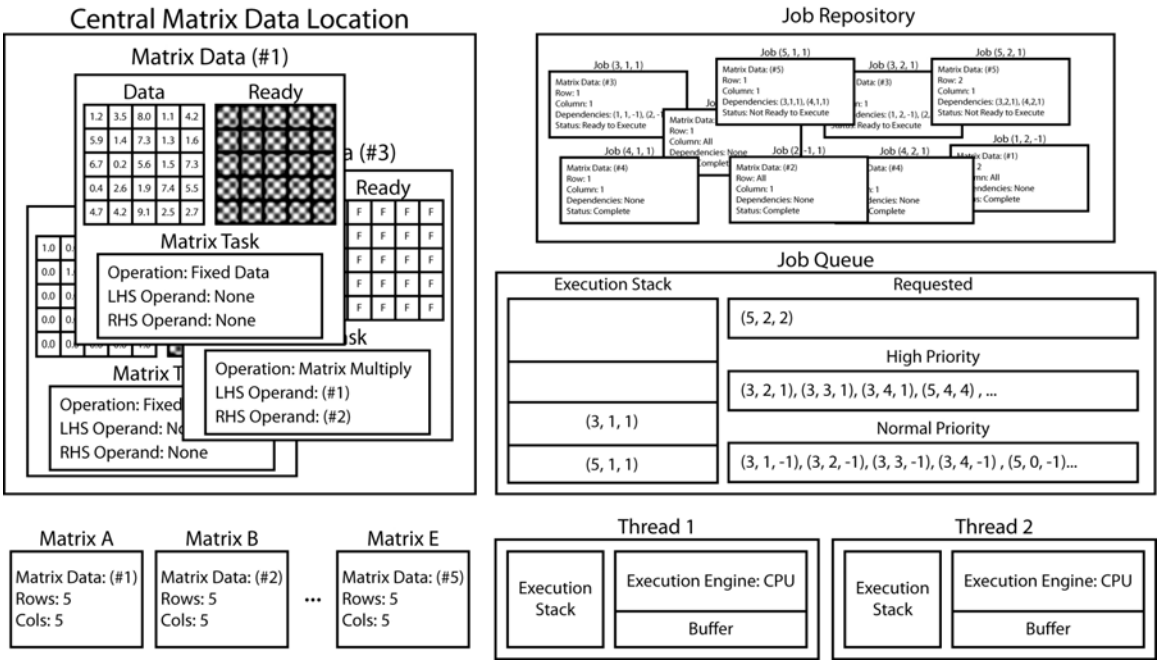

Figure 1: Overall Architecture of Asynchronous Matrix Framework

The matrix framework is a structured to make the interfaces into the framework very natural and minimize the amount of work that is necessary to set up the asynchronous part of the system. In addition to providing natural interfaces, cross-platform threading details are also handled by the framework. This framework has been implemented and tested on Windows XP, Mac OS X 10.4.8, and Fedora Core 5.

Matrices and thread creation and deletion are the interfaces into the asynchronous matrix framework while the rest of the architecture is internal to the framework. The matrix operations that are currently defined by this framework are:

- matrix multiply
- matrix add and subtract
- matrix/scalar multiply and divide.

In addition, a user can designate that a certain part of the matrix is a higher priority than the other parts of the matrix. Apart from performing operations, getting data out of the matrix, and setting priorities for different sections of the matrix, there are not any additional interfaces for the matrices.

One of the major components of our system architecture is that there is an easy way to specify the number of threads that the matrix should create at runtime. Each thread handles execution for a number of sections of the matrix computation, for example, a row of a matrix addition. The basic execution path that the threads follow is to get a section of the matrix that is not finished and execute that section. The details of multi-threading are discussed further in section 3.4. In order to set up the threads initially, the matrix framework is told how many threads to create and what execution engines each thread will use. The ***execution engine*** handles the actual computation of specific matrix operations. In this project, a standard CPU engine and a GPU engine based off of the stream processing project BrookGPU [7] show how different execution engines can be implemented. See Section 3.5 for more details. When the matrix framework is no longer needed, a single call terminates all the threads and frees all the used resources.

The ***central matrix data location*** is where all the shared matrix data is stored. A central matrix data location is used because when an operation like **C**=**A***B** is performed, the system decomposes this into the task where the data that is in **A** and **B,** at the moment

the instruction is run, should be multiplied together and stored in **C**. If the next step in the program is to set **A** = **B**, **C** is still expected to be **A***B**, not **B***B**. Because of this, the data is separated from the matrix containing it because the data in the matrix is not guaranteed to be stable over the duration of the computation. ***Matrix data*** is the expected numerical data that would be found in a synchronous matrix plus the addition of Booleans which represent whether a particular element in the matrix is ready and the task that was used to make this matrix. Matrix data is incrementally numbered when it is first added to the central matrix data location so that it can be easily identified and referenced. Matrix data is created once and referenced by any matrices or matrix tasks that use it.

A ***matrix task*** is stored by the matrix data and has the information about what operation and operands were used to create a matrix. The operands are other matrix data stored in the central matrix data location to ensure persistence of the data, or, in the case of a scalar operation, the value representing the scalar. Using the **C**=**A***B** example again, **A**'s data would be matrix data (#1) and **B**'s data would be matrix data (#2), so the matrix task would be:

1) the operation: matrix multiply
2) the left operand: (#1)
3) the right operand: (#2)

Jobs are at the heart of the asynchronous matrix framework and can be fundamentally thought of as the matrix task applied to a smaller section of the matrix. A job can represent an element, row, or column of the resulting matrix and contains some additional information as to its stage of execution and what other jobs it is dependent on before executing in order to facilitate multi-threading. Using the **C**=**A***B** example again,

C's data would be (#3), so a job that would finish row one and column one in (#3) would perform a dot product between row one of **A**'s data, (#1), and column one of **B**'s data, (#2). Without multi-threading, jobs could have only three stages of execution: not ready to execute, ready to execute, and complete. However, because multi-threads are trying to execute jobs simultaneously, the job needs another stage of execution to specify that it is being executed to prevent multiple threads from trying to execute the same job.

Jobs are stored in a ***Job Repository*** so that there is a single creation point to co-ordinate multiple threads trying to potentially get the same job and to allow jobs to be referenced multiple times. This is useful when tracking dependencies such that if **C**=**A***B**, where the elements of **A** are dependant on another computation, all the elements in a row of **C** would reference the job for that row in **A**. As the job in **A** is completed, all the elements in **C** would recognize that one of their dependencies was eliminated.

The ***Job Queue*** is a set of three queues and a stack used to enforce priorities and dependencies. Priorities are enforced by putting jobs into the queues that correspond to their priority and getting jobs out of the queues from the highest priority to the lowest priority queue. Before a job is given to the execution engine it is moved to that thread's execution stack and all the dependencies of that job are moved on top of it in the stack. By expanding out the dependencies of all jobs of the stack, the job at the top of the stack will be executed before the jobs that are dependent on it lower down the stack. The execution stack in the job queue is used if the main thread of execution requests an element of a matrix and begins to process jobs in order to finish that element and each thread has its own execution stack for when it acquires a job from one of the queues.

We detail the fundamental architecture and timings of the implementation in the

following sections. Overall, we found that, even with the added overhead required, this asynchronous matrix framework met all of our project goals. The system performs comparably to the synchronous system with a constant linear overhead that is dependent on matrix size. The system allows access to individual or groups of matrix elements before the entire matrix has been computed and much sooner than a synchronous system would allow. Finally, the asynchronous system outperforms the synchronous matrix as soon as two threads are used and increases performance by up to 70% using two threads in our testing.

CHAPTER 2

Related Work

While there are some asynchronous matrices already [15], these are talking about hardware matrices that would require a new processor instead of a software framework that runs on current microprocessors. In fact, there do not appear to be any matrix frameworks that try to delay computation possibly indefinitely while tracking dependencies. Many matrix frameworks take the approach of trying to structure computations and memory accesses in order to try to reuse data that is likely to still be in memory and take advantage of the hardware cache hierarchy [1][12] [13]. The recursive blocking algorithm in [9] is similar in concept as jobs are structured recursively but instead of recursion within a matrix the jobs span across multiple matrices and produce a single result. The use of jobs to specify sub-jobs discussed at the end of Section 3.3 is even more closely related to [9]. Another framework [5] attempted to optimize performance of a matrix multiply using code that compilers can optimize well plus having an automatically tuning block size.

A collection of linear algebra subroutines that is implemented using matrix operations is ScaLAPACK [6]. This package is a multi-threaded matrix framework that is based off of LAPACK [3], which is a linear algebra library and is designed to be scalable to distributed memory machines. This distributed memory is somewhat similar to GPU computation as the memory space for the CPU and GPU are not shared. Similar issues about communication and the amount of data that is moved from one memory space to another are encountered because the asynchronous framework allows for execution to be

implemented on many different devices.

The strategy pattern [10], which allows an algorithm to be encapsulated an a object and allows algorithms to be interchangeable, is used to enable many different implementations for execution including GPU execution. Performing matrix computation on the GPU [11] exploits the huge amount of parallel resource that GPUs contain. General purpose computing on the GPU was popularized when the Stanford Graphics Lab released BrookGPU [7], which enabled computation to be specified without knowing all the details of the graphics card. With the current generation of GPUs, both NVIDIA with CUDA [14] and AMD/ATI with their stream processing initiative [2] have put more native support for general purpose computing into their graphics cards. Previously results for matrix computation on the GPU has been very good but the difference in this paper is that computation is element independent meaning that a single element could be computed without computing any other elements. This could cause the GPU to no longer be a good fit with how the data is represented which is discussed in Section 4.3.

CHAPTER 3

Implementation

Matrix Operation Syntax	Multiply **A*B**	Addition **A+B**	Subtraction **A–B**	Scalar Multiply **A***(scalar)	Scalar Divide **A**/(scalar)

Table 1: Currently Supported Operations

Using asynchronous matrices via the matrix framework hides almost all of the differences between synchronous and asynchronous matrices from the user. The matrix class is the outermost view of an asynchronous matrix and allows for simple operators such as multiplication and addition to be defined in a clean way without having to account for any asynchronous behavior. Matrix operators that are supported at this time are shown in table 1 in addition to any chain of these operators that are meaningful. This is shown in the following pseudo code snippet:

```
Matrix A(4,3), B(3,4), C(4,4), D(3,3); // 4x3, 3x4, 4x4, 3x3 Matrix
Matrix E = A * B;             // results in a 4x4 matrix
Matrix F = B * A + D;         // results in a 3x3 Matrix
Matrix G = A * B + C * A;     // results in a 4x3 Matrix
Matrix H = A * B + D;         // runtime error, can't add 4x4 to 3x3
```

The only place where asynchronous operation causes a different interface for the user is in the matrix framework being able to initialize a number of threads to perform calculations and the ability to set a priority level for a single matrix element, row, or column. The asynchronous framework extensions allow matrices to be executed in many different ways such as executing an element only when it is asked for, executing elements in the background if there are additional threads created, or if executing elements in a different order than they were specified if they were set to a higher priority. Threads are initialized in a similar fashion to the following pseudo code:

```
Matrix.initialize( 2 CPU threads, 1 GPU thread );
```

Matrices can only be created through being filled directly with numbers or through operations between matrices. Asynchronous matrix operations can be executed between any two compatible matrices that are either filled with data directly or represent a previous matrix computation. A matrix operation is compatible if **A** and **B** have the same number of rows and columns for matrix addition and subtraction and that **A** has the same number of columns as **B** has rows for matrix multiplication. If two compatible matrices **A** and **B** were defined, **A*****B** would create an intermediary matrix that represents **A*****B** but would not perform any calculations yet. Because **A** and **B** could be assigned different values at some later time, the data that is currently stored in **A** and **B** are stored in a central location. The framework does not assume the data in the matrices to be static. By using a shared location, asynchronous operation can be performed even when the initial matrix does not persistently point to the same data throughout the calculation.

When the resulting matrix from **A*****B** is created, storage for all the elements' data is created along with a Boolean for each element to denote whether or not each data element is done. Depending on the compiler and optimizations for storing Booleans in an array as little as 1-bit per Boolean overhead is required for this additional state information. The worst case would require 32-bits per Boolean on a 32-bit system. In addition to data storage and storage to determine if each element is complete, a matrix task that specifies the operation and operands is created. For the example of **A*****B**, this would be a matrix multiply with operands of **A**'s data and **B**'s data. The final step in the matrix creation process is the creation of a number of jobs that, if all completed, would

fill in the data of the newly created matrix.

A matrix task is a data structure used by the framework to signify what operation should be performed and points to the correct operands in the shared matrix location. A task is the basic guide to creating jobs that will be able to calculate certain parts of the new matrix. Based on the task, the location of dependent elements in previous matrices is also determined. The task also is used when executing a job to tell the execution engine what operation is being performed. For each given operation, a different action in the execution engine is performed, such as with matrix multiply specifying that for an element the dot product of a row and a column needs to happen or with a matrix/scalar multiply that the element is multiplied by the scalar stored in the task.

Jobs are created that represent an element, row, or column which need some type of matrix operation executed to complete a section of the matrix according to the matrix task that is specified. When a matrix operation occurs, jobs are added into the job queue and are executed either because of being a dependency of a different job or due to being the next job out of the queue. Job dependencies are further explained in Section 3.3.

The only time that a matrix computation is not immediately able to execute is when a matrix element is requested that has dependencies that are not yet complete. This will occur only with chained matrix operations. If an element is ready, then the value can be returned immediately and no delay is experienced. If an element is not ready, the job repository is asked for a job that matches the section of the matrix that needs to be completed. That job is then put onto the execution stack and its dependent jobs likewise pushed on top of the stack. All the jobs on the execution stack are executed until the job fills in the section of the matrix that was initially requested. When this job is complete,

the value is returned as it would be in any other matrix framework.

Job execution is handled by an execution engine. The execution engine handles the actual computation of specific matrix operations. The execution engine is not tightly coupled with the other aspects of the matrix framework, and many different implementations are possible including a CPU and GPU engine for this project. As long as the execution engine has an equivalent mathematical operation defined (such as multiply), it will be able to execute that type of job. As newer technologies become available other engines that might be better suited for execution of matrices can be developed.

Overall, a job is the most vital building block of out asynchronous matrices framework. This is due to the fact that jobs allow flexible asynchronous execution to occur by providing a standard way to specify that a job is responsible for completing the necessary computation for a certain part of the matrix. In order to enable flexible execution, jobs provide a standard way of presenting data to an execution engine to ensure that they can easily process the data coming in, which is explained further in Section 3.5. Jobs are also responsible for returning values to the matrix data and marking the Booleans that specify that elements are ready (i.e. completed).

3.1 Matrix Job

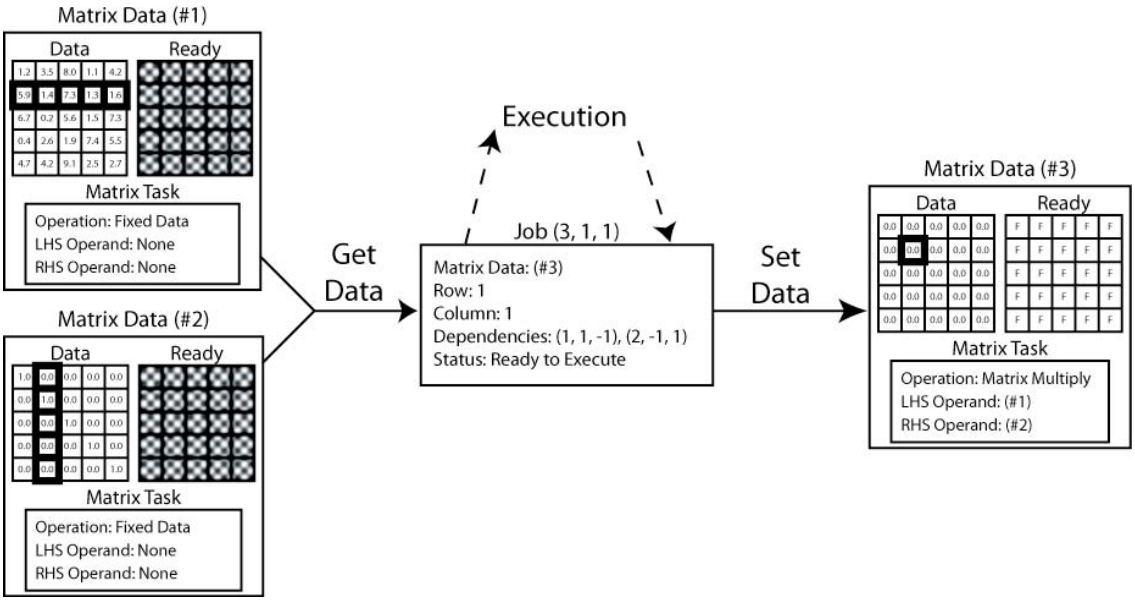

Figure 2: A job getting data from (#1) and (#2) to fill in an element of (#3).

A job defines a section of data that can be completed in a destination matrix if this job is executed. A job contains:

- its status
- where the results of the job will be stored
- the rows and columns this job defines,
- and the jobs that it is dependent on to execute.

The status of a job can be various values based on where it is in the execution process. The possible values are:

- not ready to execute,
- ready to execute,
- executing,
- and complete.

When a job is first created, it starts in the “not ready to execute” state. If a job is the next

in the queue or about to execute, it is queried to check if it is ready to execute. This, in turn, causes the job to check its dependencies and if all of the dependent jobs are complete, this job is ready to execute and transitions into the "ready to execute" state. Dependent jobs are determined by figuring out what sections of operand matrices would need to be completed to allow the current job to execute and adding jobs representing those sections to the current job. This is further explained in Section 3.3.

When a job is passed to the execution engine to begin execution, the job is moved into the executing state so that it is not given to multiple threads. After the job finishes executing and returns the values to the matrix, the job is complete and all the data that is defined by this job has been calculated and the associated Booleans are set for these elements.

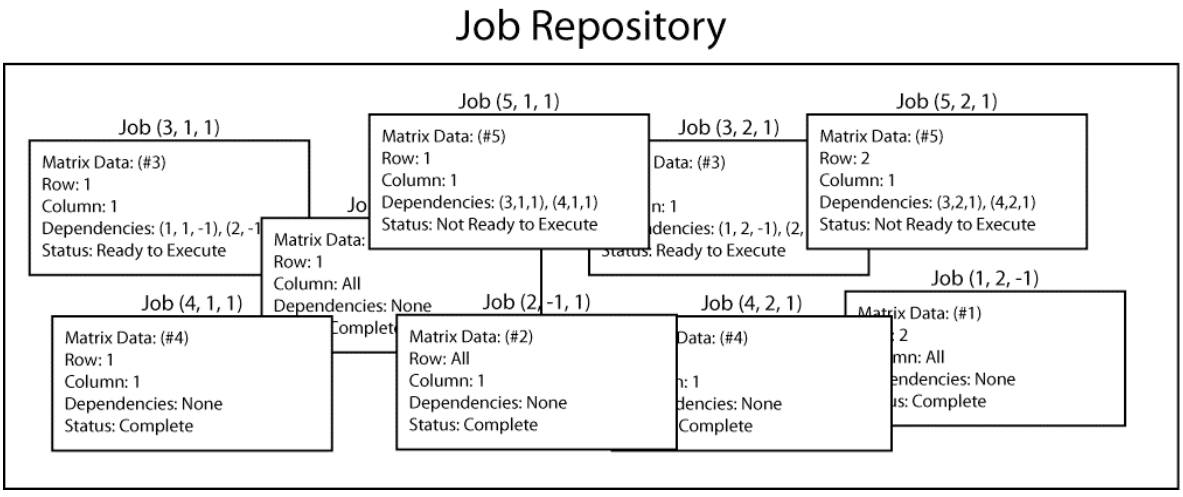

Figure 3: Job Repository stores and creates jobs.

Jobs specify what matrix data they will write into and what row and column they represent. Jobs are identified in this paper as a tuple, a group of three numbers: the matrix data to write to, the row, and the column denoted as (#, #, #). For example in matrix data (#3), row 1, column 1 would correspond to Job (3, 1, 1). The job repository

is a data structure that uses a map with the previously specified tuple used as the key which jobs are stored against. As stated earlier, the matrix data is numbered when it is added to the central matrix data location, so matrix data (#3) would be the third matrix specified. A job can also specify all rows or all columns, meaning that a column or row can be represented with a single job which is denoted as: "–1". Theoretically, the entire matrix could be represented but this was never found to be useful in the test cases explored in this thesis. Because jobs can be uniquely identified with their tuple data, jobs are only created once and, if requested again, the same job will be used in every applicable scenario. In other words, since matrix data (#3) represents a matrix multiply between (#1) and (#2), Jobs (3, 1, 0), (3, 1, 1)… (3, 1, 5) would all depend on Job (1, 1, -1) so it is created once by the job repository and used as a dependency by all the jobs in row one of matrix data (#3). Reusing jobs is also useful to eliminate recalculation of different parts of a matrix.

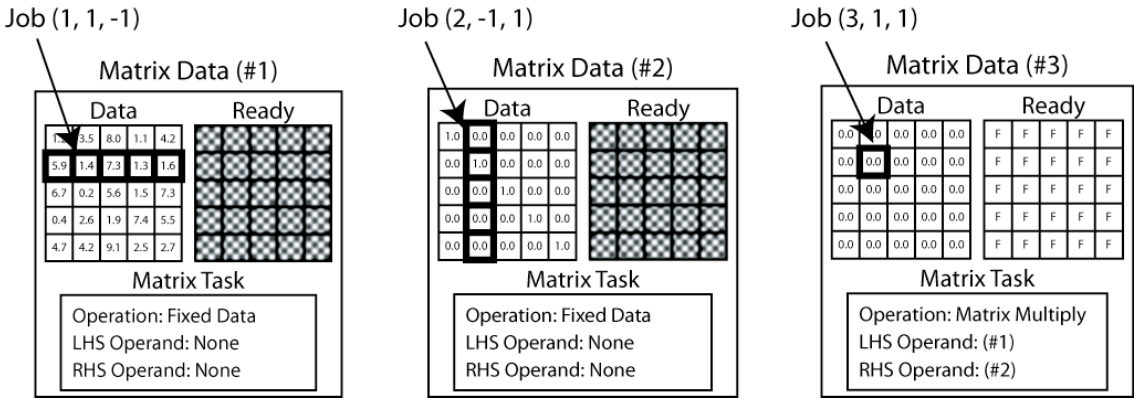

Figure 4: Jobs related to the sections of the matrix they represent.

Default jobs are created when each new matrix operation is defined. Creating individual jobs for each matrix element when a matrix operation is specified is avoided because this causes a large amount of overhead. This is due to the fact that a majority of the jobs will not have to be created individually and will only be created if a single

element is requested. However, default jobs are created and grouped together by row in order to minimize the job creation overhead while having jobs that would complete the entire matrix if executed.

Jobs get data required for execution from the operand matrices based on what task it is supposed to perform, such as matrix multiply. For example for Job (3, 1, 1), the execution data that would be required is the first row of matrix data (#1) and the first column of matrix data (#2). Based on the type of task and the section the job represents in the destination matrix, the job determines what the operand data passed onto the engine will be, whether scalar or an array of data. The job also specifies how much data will be calculated, as in a row, column, or single element.

When the execution engine has completed the calculations, the results are passed back to the job to write into the destination matrix. The data is set in the destination matrix as well as marking that chunk of the matrix as complete. After this process is finished, the job is complete and its status is updated.

In order to support asynchronous operations, a priority can be set to allow earlier execution than default jobs that were added at the lowest priority, which is explained further in the next section.

3.2 Job Priorities

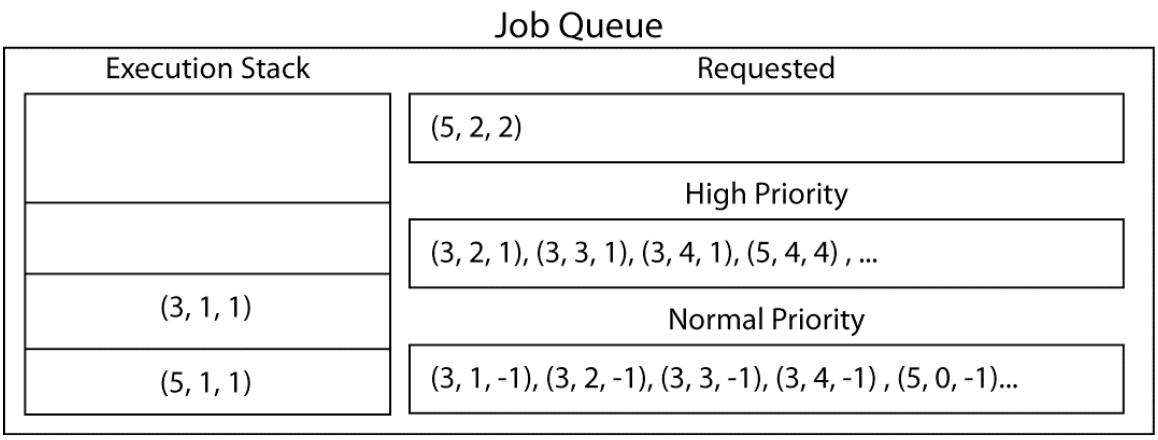

Figure 5: Job Queue enforces priorities and gives a central location for threads to acquire new jobs.

Priorities are one of the only places where asynchronous matrices look different from synchronous matrices. Different priority values correspond to the queues in the job queue being Normal, High, and Requested. To set a priority, all that is needed is to select a certain section of the matrix and specify what the new priority is. Priorities allow for individual elements or specific sections of a matrix to be prioritized to execute sooner than they would normally execute and thus they allow for asynchronous matrix computation to work. When a priority on a matrix section is set, a corresponding job from the job repository is created or requested. If the job is not already complete or executing, it is moved to the start of the queue that corresponds to the given priority. Therefore, if row 1 in a matrix was set to high priority, the job for row 1 would be put at the end of the high priority queue.

Priorities are used to determine what jobs will execute next when a thread is ready to execute another job. When a matrix element is requested, a job moves to the end of the requested priority queue and quickly moves on to execution. Priorities are useful when there are threads performing background computation of a matrix because if a

certain matrix element or section in a matrix is known to be needed sooner than the rest of the matrix, it can be prioritized so that background threads can perform those jobs first instead of blindly picking the earliest defined default jobs.

Jobs that are defined to be at the same priority level are executed in a first-in-first-out order of operation. When a matrix operation is created, it in turn creates default jobs that are needed to complete this computation and all the jobs are added to the normal priority level queue. If another matrix, created after the first operation, also makes default jobs, these would be executed after the first matrix unless priorities on those jobs were changed.

Priorities also allow jobs that are moved to a higher priority to be grouped in a different way than the default grouping which is by row. Jobs can be grouped as individual elements, as a row, as a column, or as an entire matrix. This allows either certain elements that would normally be grouped with many more elements to be calculated individually or allows an entire row or column to be given higher priority.

When a job of any priority, even the highest priority, is about to be executed, all of the jobs that it is dependent on are expanded onto the execution stack above the job. This only happens with chained operations as using fixed data there would be no dependencies. In this sense, even though the other jobs might not have been explicitly set to a certain priority level, they inherit the priority level of the job that needs them to complete.

3.3 Job Dependencies

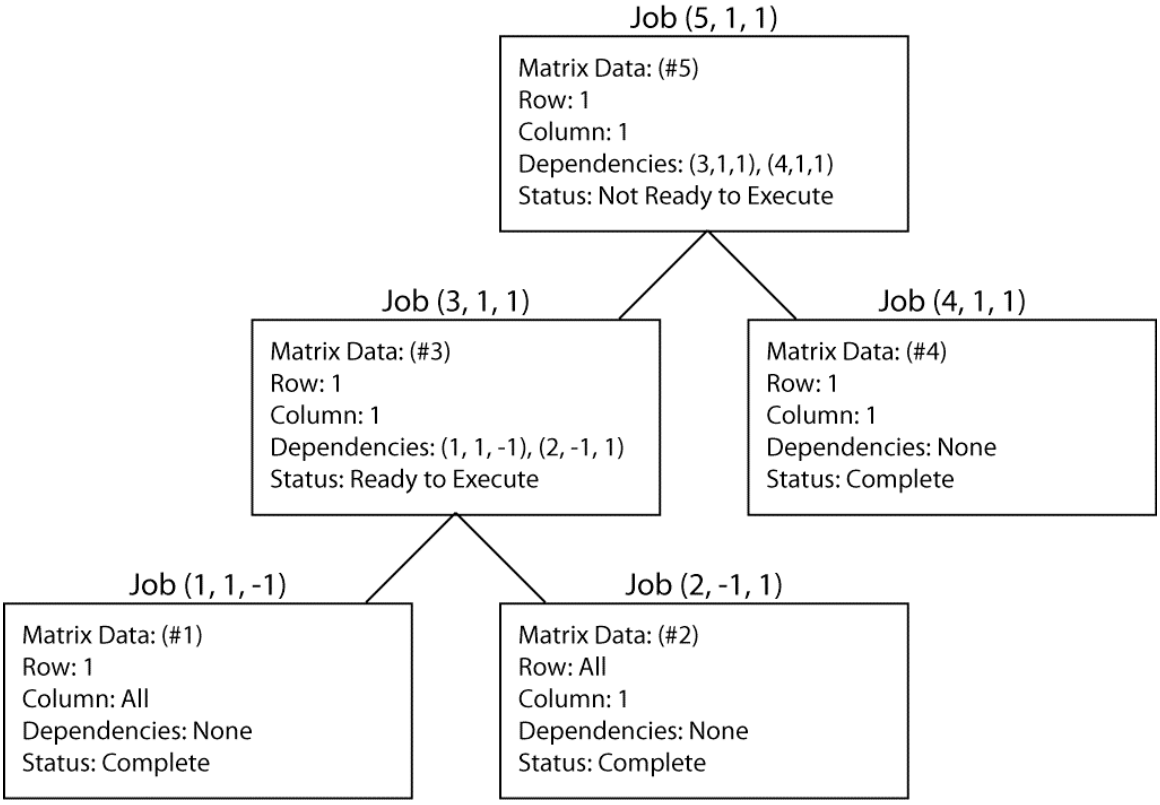

Figure 6: Job dependencies allow for the specification of multiple operations before any data is computed.

Dependencies are created based on matrix operands and the type of operation specified. If both operands were loaded directly with numbers, there would never be a case where the resulting matrix would be dependent on any parts of the previous matrices. The job would still have dependencies but those dependent jobs would always be complete. Instead of using only matrices that are loaded directly with numbers, consider if the following two sequential matrix operations were specified in an operation chain:

1) **C**=**A***B**
2) **E**=**C**+**D**

The data in **C** would not be computed when **E** is specified and therefore there is an

obvious dependence on the data of **C** to complete before data in **E** can finish. This scenario is illustrated in figure 6.

The ability to create dependencies allows a matrix operation to be defined even before the input matrices are complete. If this were not allowed, matrices that were used as inputs would have to complete *all* of their computation before returning the matrix. This would end up making it seem like all matrix operations are synchronous except for the last operations that were performed, nullifying the point of asynchronous operation and introducing the bookkeeping overhead for no good reason.

Dependencies are built into a job and basically specify what other jobs need to be completed before this job can be executed. By arranging it in this manner, one job can be dependent on another job that can in turn be dependent on other jobs. This allows a dependence chain to be built and a series of tasks that need to be completed to make the current job ready to execute. This also allows for the smallest amount of work needed to ready a job for execution to be done and is one reason why a single element can be obtained much faster than the calculations for a synchronous matrix. This is evident in figure 6 because only a dot product between row 1 in (#1) and column 1 in (#2) and the addition of that and element (1,1) in (#4) give the result for (5, 1, 1).

Dependencies are not created until a job is checked if it is ready to execute in order to speed up matrix creation time and reduce overhead. When a matrix computation is specified, there is no reason for the dependencies of each job in that matrix to be found because the jobs may not need to be executed in their default state and could possibly never be requested. Creating objects representing every possible dependency for a matrix would be far slower than synchronously performing the matrix calculations and would be

a huge space overhead.

Dependencies can also be made in order to group elements together in a job. An example of this is grouping the jobs that make up a row together. Instead of Job (3, 1, -1) depending on Job (1, 1, -1) and Job (2, -1, -1), Job (3, 1, -1) can be dependent on Jobs (3, 1, 0), (3, 1, 1)…(3, 1, n) which are then dependent on the row in (#1) and the columns in (#2). This is useful for grouping jobs from matrix operations such as a row of a matrix multiply. Instead of being dependent on the entire previous matrix to execute, each element in the row can execute as soon as the rows and columns that it depends on are finished.

3.4 Multi-Threading

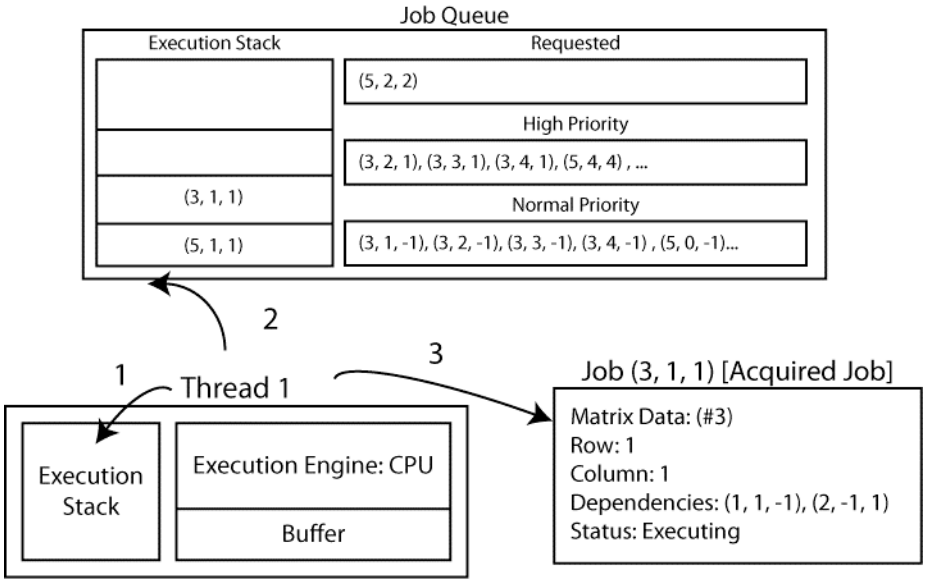

Figure 7: General multi-threading execution path

Multi-threading in this application allows for background processing of matrix elements while the main program that uses this framework continues to execute something else, for example rendering in a graphical program. The general multi-threading strategy is to have a number of threads using the following execution path:

1. Check if there are any jobs that belong to this thread.
2. If not, try to get a job from the Job Queue.
3. If a job is acquired in steps 1 or 2, execute that job with the execution engine associated with this thread.
4. Repeat until notified to terminate.

In order to enable multi-threading without race conditions, the code uses locks for job creation, setting job dependencies, adding and removing jobs from the job queue, and when creating shared storage for the matrices. If this were a completely single threaded project, all locks could be removed to eliminate any extra overhead that they may create.

This is a simple use of multi-threading and allows for easy creation of multiple computation threads to complete jobs based on the physical systems that are being used and the requirements of the application. This simple multi-threading could be a big help for an application that is using matrices as the main computational tool but is not leveraging the extra computational resources on a multi-core/processor machine. While this benefits people that have not employed multi-threading in their projects up to this point in their applications, care needs to be taken that the number of threads does not slow down the application's other functions. When using this matrix framework, we would suggest that a good number of threads to have running would equal to the number of processing cores that are available or less. If the number of threads gets too large, the threads will spend time contending with each other for resources instead of getting more meaningful work done, which is explored in Section 4.4.

3.5 Flexible Execution Engine

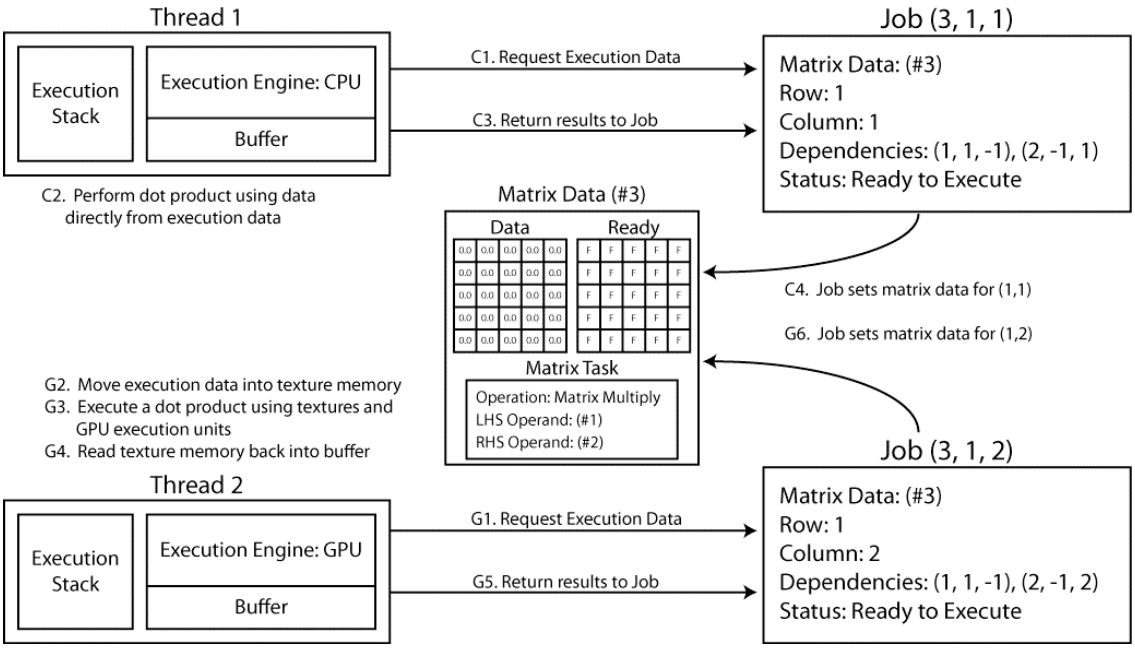

Figure 8: Job execution using CPU and GPU execution engines

Multiple execution engines are possible because the data that is retrieved from a job is always in a uniform format. The data has the matrix operation, which could be matrix multiplication, addition, etc., and arrays of data or an array and a scalar depending of the operation. By making the jobs convert execution data to a uniform format, the execution portion of the matrix is decoupled from more intimate knowledge of the jobs and allows for easier updates to include new operations and different strategies to be implemented.

Currently basic CPU and GPU execution engines are implemented to show how execution can be accomplished on two completely different processors and to measure performance using different strategies. The CPU engine is written in plain C++ using basic matrix algorithms without trying to add additional optimizations that could possibly improve performance using architecture specific vector instructions such as SSE or other

instruction set extensions. The GPU engine is written using BrookGPU [7] as the interface for general purpose computing on the GPU. In addition to the CPU and GPU engines, others could be implemented to try to collect jobs that are adjacent and group those jobs together or try to optimize other parts of the execution.

Future hardware that is optimized for stream processing, large dot products, and other operations that could be useful for matrix calculations can be implemented very easily. Any C++ class that implements the ExecutionEngine class, meaning it is capable of executing a job, knowing how to interact with a job to retrieve the necessary data for computation and returning completed results, could be an execution engine.

CHAPTER 4

Results

Various matrix sizes, operations, and threading models have been tested using our matrix framework in order to show:

- what overhead is inherent to the asynchronous design,
- how well matrix operations scale on multi-core systems,
- and different strengths and weaknesses of this framework.

Testing was performed on three different physical systems, using three different operating systems. All single core testing was performed on a system running Windows XP, dual core testing was performed on a system running Mac OS X 10.4.8, and quad core testing was performed on a system running Fedora Core 5. Testing on these different platforms shows that this framework is a portable solution for asynchronous matrix processing. In order to simplify testing and be able to easily chain together any number of operations, all matrices are square matrices. Non-square matrices have been tested and are supported on this framework, but their timings are not included, because the timings did not show any interesting differences and non-square matrices are more difficult to string together in order to test longer operation sequences.

Three different types of execution are tested:

- Synchronous matrix execution: completes the entire matrix then returns. Used as the baseline for comparisons for other execution types.
- Just-in-time execution: does not create any additional threads and only does work when an element from the matrix is requested.
- Asynchronous execution: creates one or more threads to do work in the

background in addition to using the just-in-time idiom when an element is requested.

Unless otherwise noted, all timings are in seconds and all matrix elements are computed. The following results include asynchronous results for more than one thread to present how this framework scales but direct comparisons should be drawn only between synchronous and single threads.

4.1 Overhead

In order to reveal the overhead introduced by moving from a synchronous execution to an asynchronous model, synchronous, just-in-time, and asynchronous matrix operations were all tested using one thread of execution with matrices of various sizes.

Size	Synchronous	Just In Time	Asynchronous (1 thread)
1,000 x 1,000	3	5	5
2,000 x 2,000	20	29	31
5,000 x 5,000	303	375	374

Table 2: A*B+C (seconds)

Using one thread of execution, it is not surprising that there is an increase in overhead. There are many different reasons why this overhead accumulates from the asynchronous framework. The synchronous version does not have to keep any extra objects or other data that is associated with completing parts of the matrix after the initial matrix operations are defined. Some of the sources of overhead come from creating jobs and tracking the dependencies between jobs in order to ensure asynchronous operation. Another reason overhead is introduced is due to the corresponding array which keeps track of what elements have been completed in the matrix data.

While the synchronous matrix scales almost exactly with the increase in area and execution complexity, where a doubling in size should increase by eight times and an increase of two and half should increase by 15.625, the asynchronous matrices scale better moving to larger sizes. The overhead has less of an impact with the asynchronous version increasing only 6.2 times when doubling and 12.1 times with a 2.5 increase. This shows that the overhead is not completely dependent on the size of the matrix.

When comparing the just-in-time and the asynchronous, the differences in speed are not due to object creation or bookkeeping because the same amount of work is being done. Instead, context switches between the main thread and the asynchronous thread and contention for execution resources introduces enough variability to explain the result. Because the results are so similar and because the just-in-time solution is not scalable, the rest of the testing is performed using synchronous and asynchronous matrices.

4.2 Operations

The type of operation and size of matrices affects the time that an operation will take to complete. Consider comparing a matrix multiply to a matrix addition, instead of just a single addition per element, for a matrix multiply there are **n** multiplications and **n-1** additions where **n** is the inner matrix dimensions. Obviously, the multiplication is much slower and testing was done to determine how much each operation contributes to the overall execution time. Also examined were how well matrix operations scale when strung together and the effects of different size matrices on how long matrix computations take to execute.

4.2.1 Individual Operation Comparison

	Synchronous	Asynchronous (1 thread)	Asynchronous (2 threads)
Matrix Multiply	3586	3652	2101
Matrix Addition	4	6	6
Matrix Subtraction	4	6	6
Matrix Scalar Multiplicaiton	0*	6	6
Matrix Scalar Division	0*	6	6

Table 3: All operations performed on 10,000 x 10,000 matrices (seconds)

***operations took less than 1 second**

Using significantly larger matrices allowed operations to be tested individually to see how much time each type of operation takes. Until matrix sizes are well above 5,000, performing a synchronous matrix addition, subtraction or any scalar-matrix operation takes less then one second. In fact, a synchronous matrix performing a scalar-matrix operation on a 10,000 x 10,000 matrix still finishes its operation in less than one second. In terms of execution power differences in scalar-matrix and matrix operations, excluding multiplication, the amount of computation is about the same between a single add, subtract, multiply, or divide per element. Matrix multiplication on a matrix of this size uses approximately 10,000 times as many multiplies and adds per element.

One might ask, if scalar-matrix and non-multiply matrix operations perform equal amounts of arithmetic operations, why are the scalar-matrix operations so much faster on the synchronous version? One possible explanation is that the matrix operations have to work with twice the amount of data and pull that data into the processor before it can

perform its work. This could be one reason that the matrix multiply does not take a full 10,000 times longer to execute compared to the other matrix operations since parts of the matrix are fresh in memory and do not have to be retrieved again. Our testing did not account for any possible caching of matrix data elements between operations.

Asynchronous matrices have a fixed overhead, no matter which operation, as jobs need to be made to ensure the matrix completes and need to check if certain parts of the matrix are completed. This can be seen in the matrix-scalar operation compared to the simple matrix operations as the times are all exactly the same. Because the majority of the execution time for non-multiply operations in the asynchronous version is spent fetching data to perform operations, adding additional threads does not help in those cases and as seen above, slightly slows down computation as there is contention for memory and synchronization objects in the program. The matrix multiply performed on large matrices shows a very good speed up when more threads are allowed to execute as opposed to the simpler operations.

4.2.2 Composite Operations (Matrix Multiplication Only)

	Synchronous	Asynchronous	
		(1 thread)	(2 threads)
A*B	29	38	21
A*B*C	57	75	42
A*B*C*D	85	114	62
A*B*C*D*E	113	154	84
Average per Multiply	28.4	38.1	20.9

Table 4: Multiplication scaling on 2,000 x 2,000 matrices (seconds)

When chaining multiplication operations, our asynchronous framework always performs faster then synchronous when using two threads. When performing multiple

operations on matrices there is a memory overhead and an execution overhead. When stringing matrix multiplies together the typical amount of time for execution of a single operation can be seen. Using a single thread of execution, it is always faster to complete the entire matrix with the synchronous version. The matrix multiplications are not dependent on the total number of operations that have been issued but only on the size and for the asynchronous version, the amount of bookkeeping overhead and synchronization that is happening. For 2,000 x 2,000 matrices, the amount of overhead that is added to each operation is an additional 30% compared to the synchronous version for matrix multiplies. Using two threads, if the threading was completely independent and there were no additional overhead because of thread interaction, the total execution time would drop to 66% of the synchronous time. Instead because of the interaction between just two threads the total execution time is about 72% of the synchronous version.

4.2.3 Composite Operations (Various Operands)

	Synchronous	Asynchronous	
		(1 thread)	(2 threads)
A*B+C*D-E	57	76	42

Table 5: Composite operations performed on 2,000 x 2,000 matrices (seconds)

This test uses the same size matrices as before but adds in additional matrices being added and subtracted. The additions and subtractions have no appreciable effect on the total execution time compared to the same operation without those operations. We again see that our asynchronous framework with two threads performs faster then the synchronous. This is not entirely surprising as the number of additions is comparable to the combined additions and multiplications in half of one row of a matrix multiplication.

This test also shows that creating the additional storage to house the intermediate matrices and create jobs for additions and subtractions is trivial compared to matrix multiplication.

4.2.4 Composite Operations, Smaller Matrices (Various Operands)

	Synchronous	Asynchronous	
		(1 thread)	(2 threads)
A*B+C*D-E*F+G*H-I	14	26	14

Table 6: Composite operations performed on 1,000 x 1,000 matrices (seconds)

This test strings together many smaller matrices to see how size and number of matrices affects the framework. Because of the overhead that the asynchronous matrices introduce per job compared to the total amount of execution to be done to complete the matrix is higher per element, the asynchronous matrices do not perform as well on smaller matrices. Likewise the reverse is true; larger matrix operations have a lower amount of overhead compared to computational work so they perform better relative to the synchronous matrices. Threads can make up for some of the overhead when using asynchronous matrices but with sufficiently small matrices the threads cannot make up for the overhead.

4.2.5 Same Operation on Different Size Matrices

Size	Synchronous	Asynchronous	
		(1 thread)	(2 threads)
2,000 x 2,000	29 100%	38 131%	21 72%
5,000 x 5,000	447 100%	493 110%	282 63%

Table 7: A*B+C Effects of size on relative computation time (seconds)

As shown in the above table, when the size of the matrix is smaller, the percent of

execution that is overhead compared to actual work is larger. While with size 2,000 x 2,000 matrices the overhead is about 30% as mentioned before, size 5,000 x 5,000 matrices have an additional overhead of only 10%. This shows that larger matrices are the ideal case for asynchronous matrices as the overhead is very negligible while with very small matrices the overhead can climb to 100% or more extra work. Also of note is that with all sizes, the increase in speed moving from one to two threads is between 55% and 75% with smaller matrices seeing better scaling.

4.3 GPU

This matrix framework supports multiple execution strategies and to show how this functionality works, a GPU execution strategy was implemented. Testing was done to figure out how fast this form of execution is and what the drawbacks and benefits of this approach are. This implementation was thought to be a good source of extra execution units for computation as GPU accelerated matrices have shown good performance improvements in the past.

Size	GPU Execution (seconds)	GPU Memory Reads	GPU Memory Writes
500 x 500	51	501000	250500
1,000 x 1,000	218	2002000	1001000
2,000 x 2,000	1001	8004000	4002000

Table 8: A*B+C GPU Execution

As can be seen with the above numbers, GPU execution is far slower when used with this framework instead of performing the computations on the CPU. The asynchronous matrix framework's main objective is to minimize the amount of work that is done at one time to the point where matrix multiplies compute every element in the matrix independently of all the other elements. This leads to a very poor fit for use with the

GPU at least in the current general purpose GPU climate. Because of how the matrices are implemented and their general independence between elements, there ends up being a very large number of GPU memory reads and writes which was tested in tuning suites to be the source of the long execution times. Other frameworks are able to minimize the number of GPU memory reads and writes because they can make assumptions that the matrix will go through the entire execute before being returned and that all the matrices that are being used as input are completed before being passed into another matrix operation.

With the current execution strategy used in the asynchronous matrices, the basic operations that are computed on the GPU spend far less time executing than loading data into graphics memory. This can be seen in the comparison between the CPU and GPU versions and how they scale with matrix size. The expectation is that if a resource is being fully utilized by a matrix multiply then the execution time would be eight times as much as a matrix of half the size seeing as there is a quadrupling of matrix elements and doubling of execution per element. In the CPU versions, the scaling is in the expected range increasing by about 8x but the GPU seems to scale fairly closely to the number of elements and GPU memory operations. The additional increase makes sense as accessing more memory and writing and reading larger sections of GPU memory would take a bit longer.

In the future, if graphics cards can have an open path to and from main memory instead of requiring dedicated GPU memory manipulation, the execution could benefit from the additional computational resources. In the newer graphics cards, CUDA [14] from nVidia and stream processing [2] from AMD/ATI could solve some of the problems

that currently limit this framework, though if these new execution paths that graphics venders have recently opened rely on execution from GPU memory or other expensive memory operations this framework will have to wait for different advances or change from a completely asynchronous model per element to a block based system.

4.4 Multi-threading

This framework aims to be a simple solution to exploit hardware advances to automatically increase the execution speed by utilizing all of the resources that are given to the program. The framework allows the number of threads to be chosen at runtime so a program can be tuned based on the system it is running on. Three different systems with different numbers of processor cores performed the same operations with the synchronous as a baseline and a number of threads created for the asynchronous version. The single core version was executed on Windows XP, the dual core on Mac OS X 10.4.8, and the quad core was run on Fedora Core 5.

	Synchronous	Asynchronous			
		(1 thread)	(2 threads)	(4 threads)	(8 threads)
single core	20	31	30	28	28
dual core	28	38	21	24	43
quad core	34	45	24	21	21

Table 9: A*B+C on 2,000 x 2,000 matrices using 3 different platforms (seconds)

The assumption before performing this test was that, when the number of threads of execution matched the number of physical cores, the execution time would be the lowest compared to all other possibly combinations. This was only the case on the dual core system and the other systems managed to continue to decrease execution time for at least double the number of threads compared to cores. Moving from one thread to two threads

on the dual and quad core machines decreases the total execution time compared to a single thread to 55% on the dual core and 53% on the quad core and in both cases made the execution time less than the synchronous version. For the quad core moving from two to four threads increased the speed of execution by 3 seconds and lowered the total execution time to 47% compared to one thread. This shows that synchronization starts to play a large part in terms of how much speedup can be obtained with this framework in its current state. In all multi-processor cases, when the asynchronous version is running the same number of threads as physical cores, the asynchronous version is faster than the synchronous version. Differences in physical hardware and operating system differences account for the differing timings for the single-threaded versions.

It is interesting to note that both the Windows and Fedora systems allow for a difference between the number of threads and the number of cores without any immediate loss in execution speed. With the exact same underlying implementation, the Macintosh saw an immediate and non-trivial increase in execution time as soon as the number of threads exceeded the number of physical cores in the machine. Because of this the ability to specify the number of threads at runtime could be critical on machines that exhibit the same characteristics as the Macintosh.

4.5 Partial Completion

Up until this point of the testing the only way asynchronous matrices have been able to compete is by using multiple threads, but being able to return results before the matrix is fully complete or not calculating parts of the matrix are important features of our system. Being able to process only part of a matrix and use the information in that matrix to either start other paths of execution or decide that the rest of the matrix does not

need to be finished is only possible with our asynchronous matrix framework. Testing was preformed with various amounts of the matrix requested and different numbers of threads on the quad core system.

	Synchronous	Asynchronous			
		(1 thread)	(2 threads)	(4 threads)	(8 threads)
% Complete					
25	34	12	7	6	6
50	34	24	12	11	11
75	34	35	18	16	16
100	34	45	24	21	21

Table 10: A*B+C on 2,000 x 2,000 matrix. Different completion percentages. (seconds)

As seen in table 10, the time to retrieve even a single element from the synchronous matrix is 34 seconds. The asynchronous matrices are able the retrieve close to 75% of the matrix when using a single thread in about the same amount of time and are faster for all amounts lower than that percentage or when using more threads. Using a single asynchronous thread, pulling 50% of the matrix is completed 10 seconds faster and 25% is 22 seconds faster than the synchronous version providing a 42% and 183% increase in speed respectively. The amount of time to retrieve a certain number of elements is linear compared to retrieving all of the elements showing there is not a penalty for retrieving a small number of elements from the matrix compared to the entire matrix.

Asynchronous		
(1 thread)	(2 threads)	(4 threads)
104000	218000	282000

Table 11: A*B+C on 2,000 x 2,000 matrix. Elements complete after one second

In just one second of execution, there are a number of elements that are ready and can be used to start other tasks or make decisions. One possible use could be if all the elements in the matrix need to be post processed and can be streamed to that other process as they

complete.

	Synchronous	Asynchronous			
		(1 thread)	(2 threads)	(4 threads)	(8 threads)
% Complete					
25	1067	297	164	144	143
50	1067	595	325	284	284
75	1067	890	484	424	428
100	1067	1188	640	562	564

Table 12: A*B*C on 5,000 x 5,000 matrix. Different completion percentages. (seconds)

This larger matrix, performing two matrix multiplies shows how asynchronous matrices have the advantage in every multi-threaded test and tests where 75% or less of the elements are requested. When only 25% of the elements are requested and 4 threads are used, the computation time takes only 13% of the time the synchronous matrix takes and all before even a single element is available in the synchronous matrix. When comparing synchronous to asynchronous with a single thread, pulling 90% or fewer matrix elements would be as fast or faster using the asynchronous framework. Pulling 75% of the elements only takes 83% compared to synchronous, pulling 50% takes 56% compared to synchronous, and pulling 25% takes 28% compared to synchronous. This shows how asynchronous matrices benefit even without multi-threading.

CHAPTER 5

Conclusion

Asynchronous matrices provide a framework that allows a very similar representation to how synchronous matrices would be represented and therefore allows it to be used with almost no additional knowledge on the user's part. In order to support prioritizing matrix elements, one additional function needed to be added though this is only necessary to force certain matrix sections to be completed sooner than normal. Asynchronous matrices allow for each individual element to be completed independently from all the other elements in the matrix allowing for the sections of the matrix to be completed while others have not been computed at all. Applied to a graphical simulation, only the matrix elements necessary for a given frame of a simulation can be computed quickly without waiting for the rest of the matrix elements to be done.

Multithreading is easily accomplished using this framework and can be set at runtime to match the host machine's capabilities. This allows using this framework to take advantage of parallel execution and additional processing units using a very simple programming model without the user needing to understand parallel programming. Multithreading produced between a 33% to a 85% increase in speed moving from one to two threads depending on matrix dimensions when there were two of more physical cores but the current implementation does not scale as well moving from two to four threads as it has too much high level synchronization, though this could be fixed using locking based on the matrices involved or by making a single thread responsible for job scheduling.

As matrix operations are performed on larger matrices, the execution time of the asynchronous version becomes comparable to the synchronous version. With the largest matrices tested the overhead of the asynchronous matrix was only 1.8% though with smaller matrices tested the overhead could be up to 85% additional execution time. When paired with multi-threading, the framework performed well on completing faster than the synchronous version in most cases.

A clear win of our system is its ability to compute parts of the matrix as opposed to the entire matrix. This system consistently returns 50% of the elements faster than the synchronous framework while only using a single thread. In most cases, 75% or more of the matrix elements can be computed faster than the synchronous framework depending on matrix size. When it is known that a problem will only need a subset of the elements in a matrix, the asynchronous framework provides a significant advantage over synchronous matrices.

Overall, asynchronous matrices perform well on large matrices, are able to leverage multithreading and different execution strategies as they come available, and perform faster in cases where the entire matrix is not needed. With some additional work, overhead could be reduced bringing the speed to a much more comparable level without the need for threading and threading benefits could be extended even further in a more linear manner.

CHAPTER 6

Future Work

One obvious extension to this work is to implement more execution strategies. There are numerous different strategies that could be tried such as implementing execution on a new device such as a physics card or changing how individual jobs were handled by an execution engine. Different instruction set advances in future processors could be useful such as a hardware optimized dot product. Another clear extension to enhance the framework is to implement additional matrix operations such as inversion.

Grouping jobs that are similar in order to reduce memory access by lowering how many times a set of numbers is iterated over could increase the total speed of the matrix multiply. This could be accomplished by having a row and finding other columns that are ready to execute and performing operations on all available data at the same time. Another quick extension would be to store what operands and operations occurred so that in the case where an operation like (**A*****B**) happened in multiple places, the same end matrix would be used in order to save processing where duplication is hidden or the program was computing similar matrices in different areas.

In order to increase thread throughput and decrease locking contention, finer grain locking of matrices and job storage locations could make the increase moving to more threads closer to a linear increase.

An additional extension that could be possible is to allow load sharing between threads. Currently, once a thread acquires a job, it has to execute all of that job's dependencies. It should be possible to allow threads to acquire jobs from other threads so

that if there were no other jobs available in the main job queue, a thread would be able to take a job from a thread that still had work to do.

Another possible and perhaps most promising direction is to set a minimum block size and perform all operations using blocks of data rather than individual matrix elements. While this means that there is not quite the same fine grain control over which elements execute and which do not, this lowers the overhead in checking whether parts of the matrix are ready, removes a good amount of the storage overhead, and reduces the overall job overhead. If the block sizes correspond to a good GPU memory size for graphics cards, the ability to use the GPU as a resource would be greatly enhanced as the number of GPU memory operations would drop and the amount of work per section of GPU memory would increase.

References

[1] Adams, M. D. and Wise, D. S. 2006. Seven at one stroke: results from a cache-oblivious paradigm for scalable matrix algorithms. In *Proceedings of the 2006 Workshop on Memory System Performance and Correctness* (San Jose, California, October 22 - 22, 2006). MSPC '06. ACM Press, New York, NY, 41-50.

[2] Advanced Micro Devices, Inc. “AMD Stream Computing.” 23 Apr. 2007 <http://ati.amd.com/technology/streamcomputing/index.html>.

[3] Anderson, E., Bai, Z., Dongarra, J., Greenbaum, A., McKenney, A., Du Croz, J., Hammarling, S., Demmel, J., Bischof, C., and Sorensen, D. 1990. LAPACK: a portable linear algebra library for high-performance computers. In *Proceedings of the 1990 ACM/IEEE Conference on Supercomputing* (New York, New York, November 12 - 16, 1990). Conference on High Performance Networking and Computing. IEEE Computer Society, Washington, DC, 2-11.

[4] Baraff, D. and Witkin, A. 1998. Large steps in cloth simulation. In *Proceedings of the 25th Annual Conference on Computer Graphics and interactive Techniques* SIGGRAPH '98. ACM Press, New York, NY, 43-54.

[5] Bilmes, J., Asanovic, K., Chin, C., and Demmel, J. 1997. Optimizing matrix multiply using PHiPAC: a portable, high-performance, ANSI C coding methodology. In *Proceedings of the 11th international Conference on Supercomputing* (Vienna, Austria, July 07 - 11, 1997). ICS '97. ACM Press, New York, NY, 340-347.

[6] Blackford, L. S., Choi, J., Cleary, A., D'Azeuedo, E., Demmel, J., Dhillon, I., Hammarling, S., Henry, G., Petitet, A., Stanley, K., Walker, D., and Whaley, R. C. 1997 *ScaLAPACK User's Guide*. Society for Industrial and Applied Mathematics.

[7] Buck, I., Foley, T., Horn, D., Sugerman, J., Fatahalian, K., Houston, M., and Hanrahan, P. 2004. Brook for GPUs: stream computing on graphics hardware. In *ACM SIGGRAPH 2004 Papers* (Los Angeles, California, August 08 - 12, 2004). J. Marks, Ed. SIGGRAPH '04. ACM Press, New York, NY, 777-786.

[8] Fedkiw, R., Stam, J., and Jensen, H. W. 2001. Visual simulation of smoke. In *Proceedings of the 28th Annual Conference on Computer Graphics and interactive Techniques* SIGGRAPH '01. ACM Press, New York, NY, 15-22.

[9] Frens, J. D. and Wise, D. S. 1997. Auto-blocking matrix-multiplication or tracking BLAS3 performance from source code. In *Proceedings of the Sixth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Las Vegas, Nevada, United States, June 18 - 21, 1997). M. A. Berman, Ed. PPOPP '97. ACM Press, New York, NY, 206-216.

[10] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. 1995 *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman Publishing Co., Inc.

[11] Krüger, J. and Westermann, R. 2003. Linear algebra operators for GPU implementation of numerical algorithms. In *ACM SIGGRAPH 2003 Papers* (San Diego, California, July 27 - 31, 2003). SIGGRAPH '03. ACM Press, New York, NY, 908-916.

[12] Navarro, J. J., García-Diego, E., and Herrero, J. R. 1996. Data prefetching and multilevel blocking for linear algebra operations. In *Proceedings of the 10th international Conference on Supercomputing* (Philadelphia, Pennsylvania, United States, May 25 - 28, 1996). ICS '96. ACM Press, New York, NY, 109-116.

[13] Navarro, J. J., Juan, T., and Lang, T. 1994. MOB forms: a class of multilevel block algorithms for dense linear algebra operations. In *Proceedings of the 8th international Conference on Supercomputing* (Manchester, England, July 11 - 15, 1994). ICS '94. ACM Press, New York, NY, 354-363.

[14] NVIDIA Corporation. “NVIDIA CUDA Homepage.” 17 Apr. 2007 <http://developer.nvidia.com/object/cuda.html>.

[15] Tugsinavisut, S., Hong, Y., Kim, D., Kim, K., and Beerel, P. A. 2005. Efficient asynchronous bundled-data pipelines for DCT matrix-vector multiplication. *IEEE Trans. Very Large Scale Integr. Syst.* 13, 4 (Apr. 2005), 448-461.