Runtime Data Flow Graph Scheduling of Matrix Computations with Multiple Hardware Accelerators

FLAME Working Note #50

Ernie Chan
Department of Computer Science
The University of Texas at Austin
Austin, Texas 78712
echan@cs.utexas.edu

Francisco D. Igual
Departamento de Ingeniería y Ciencia de Computadores
Universidad Jaume I
12.071–Castellón, Spain
figual@icc.uji.es

Abstract

In our previous work, we have presented a systematic methodology for parallelizing dense matrix computations using a separation of concerns between the code that implements a linear algebra algorithm and a runtime system that exploits parallelism for which only relatively simple scheduling algorithms were used to parallelize a wide range of dense matrix computations. We have extended the runtime system to utilize multiple hardware accelerators yet constrained its use to a particular scheduling algorithm. In this paper, we develop a new domain-specific scheduling algorithm that addresses load balance and data locality simultaneously, which is motivated by queueing theory. In order to apply this domain-specific scheduling algorithm for utilizing multiple hardware accelerators, we implement a new software managed cache coherency mechanism that allows for the use of any scheduling algorithm. We provide performance results that validate that our domain-specific scheduling algorithm consistently attains exceptional performance on a homogeneous multicore platform and heterogeneous platform with multiple hardware accelerators.

1 Introduction

Exploiting parallelism is becoming paramount for attaining high performance on current multicore and future many-core computer architectures [28]. As a result, handling the burgeoning complexities of these parallel computing platforms is falling upon programmers. We advocate the adoption of a clear separation of concerns between algorithms and architectures where parallelism is exploited from the inherent data flow of the computation as opposed to the explicit control flow specified by the implementation. Providing the software tools and abstractions for implementating this separation of concerns for a vast range of problem domains remains a preeminent challenge in computer science, so we focus on dense matrix computations.

We have developed a simple yet elegant methodology for parallelizing dense matrix computations on shared-memory computer architectures for which we coined the name *SuperMatrix* [18, 39]. We map a dense linear algebra operation to a directed acyclic graph (DAG) and then use out-of-order scheduling techniques, akin to superscalar computer architectures, to exploit parallelism. We have applied this methodology to parallelize many Linear Algebra PACKage (LAPACK) [4] level operations, including Cholesky factorization [20], LU factorization with partial pivoting [19] and incremental pivoting [37], QR factorization [38], and inversion of a symmetric positive definite matrix [21] along with the entire level-3 Basic Linear Algebra Subprograms (BLAS) [22]. In previous work, we have presented the general methodology for parallelizing these operations on shared-memory computer architectures while only using relatively simple scheduling algorithms.

We have also extended SuperMatrix to utilize multiple hardware accelerators [36] such as graphics processors. The key to our methodology is that we encapsulate the complexities of exploiting parallelism within a runtime system. As a result, the code that implements a dense linear algebra algorithm remains constant while the modular runtime system adapts to different computing environments whether it be homogeneous multicore platforms or heterogeneous platforms with multiple hardware accelerators. It is this separation of concerns that allows us to easily adapt to different environments with various scheduling algorithms.

The contributions of the present paper include:

$$A = \begin{pmatrix} \alpha_{0,0} & \alpha_{0,1} & \cdots & \alpha_{0,n-1} \\ \alpha_{1,0} & \alpha_{1,1} & \cdots & \alpha_{1,n-1} \\ \vdots & \vdots & \ddots & \vdots \\ \alpha_{n-1,0} & \alpha_{n-1,0} & \cdots & \alpha_{n-1,n-1} \end{pmatrix}$$

$$L = \begin{pmatrix} \lambda_{0,0} & 0 & \cdots & 0 \\ \lambda_{1,0} & \lambda_{1,1} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ \lambda_{n-1,0} & \lambda_{n-1,0} & \cdots & \lambda_{n-1,n-1} \end{pmatrix}$$

for $j = 0, \cdots, n-1$ **do**
- **for** $k = 0, \cdots, j-1$ **do**
  - **for** $i = 0, \cdots, k-1$ **do**
    - $\alpha_{j,i} := \alpha_{j,i} + \alpha_{j,k}\lambda_{k,i}$
  - $\alpha_{j,k} := \alpha_{j,k}\lambda_{k,k}$
- **for** $i = 0, \cdots, j-1$ **do**
  - $\alpha_{j,i} := \alpha_{j,i} + \frac{1}{2}\alpha_{j,j}\lambda_{j,i}$
- **for** $k = 0, \cdots, j-1$ **do**
  - **for** $i = 0, \cdots, k-1$ **do**
    - $\alpha_{k,i} := \alpha_{k,i} + \alpha_{j,k}\lambda_{j,i}$
    - $\alpha_{k,i} := \alpha_{k,i} + \lambda_{j,k}\alpha_{j,i}$
  - $\alpha_{k,k} := \alpha_{k,k} + 2\alpha_{j,k}\lambda_{j,k}$
- **for** $i = 0, \cdots, j-1$ **do**
  - $\alpha_{j,i} := \alpha_{j,i} + \frac{1}{2}\alpha_{j,j}\lambda_{j,i}$
- **for** $i = 0, \cdots, j-1$ **do**
  - $\alpha_{j,i} := \lambda_{j,j}\alpha_{j,i}$
- $\alpha_{j,j} := \alpha_{j,j}\lambda_{j,j}^2$

$$A = \begin{pmatrix} A_{0,0} & A_{0,1} & \cdots & A_{0,N-1} \\ A_{1,0} & A_{1,1} & \cdots & A_{1,N-1} \\ \vdots & \vdots & \ddots & \vdots \\ A_{N-1,0} & A_{N-1,0} & \cdots & A_{N-1,N-1} \end{pmatrix}$$

$$L = \begin{pmatrix} L_{0,0} & 0 & \cdots & 0 \\ L_{1,0} & L_{1,1} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ L_{N-1,0} & L_{N-1,0} & \cdots & L_{N-1,N-1} \end{pmatrix}$$

where $A_{i,j}, L_{i,j} \in \mathbb{R}^{b \times b}$ and $N = \lceil \frac{n}{b} \rceil$ and $i, j = 0, \cdots, N-1$

for $j = 0, \cdots, N-1$ **do**
- **for** $k = 0, \cdots, j-1$ **do**
  - **for** $i = 0, \cdots, k-1$ **do**
    - $A_{j,i} := A_{j,i} + A_{j,k}L_{k,i}$ (GEMM)
  - $A_{j,k} := A_{j,k}L_{k,k}$ (TRMM)
- **for** $i = 0, \cdots, j-1$ **do**
  - $A_{j,i} := A_{j,i} + \frac{1}{2}A_{j,j}L_{j,i}$ (SYMM)
- **for** $k = 0, \cdots, j-1$ **do**
  - **for** $i = 0, \cdots, k-1$ **do**
    - $A_{k,i} := A_{k,i} + A_{j,k}^T L_{j,i}$ (GEMM)
    - $A_{k,i} := A_{k,i} + L_{j,k}^T A_{j,i}$ (GEMM)
  - $A_{k,k} := A_{k,k} + A_{j,k}^T L_{j,k} + L_{j,k}^T A_{j,k}$ (SYR2K)
- **for** $i = 0, \cdots, j-1$ **do**
  - $A_{j,i} := A_{j,i} + \frac{1}{2}A_{j,j}L_{j,i}$ (SYMM)
- **for** $i = 0, \cdots, j-1$ **do**
  - $A_{j,i} := L_{j,j}^T A_{j,i}$ (TRMM)
- $A_{j,j} := L_{j,j}^T A_{j,j} L_{j,j}$ (SYGST)

Figure 1: Unblocked algorithm and algorithm-by-blocks for the reduction from a symmetric definite generalized eigenproblem to standard form (left and right, respectively). Here a loop will not be executed if its terminal index is -1.

- A description of a new and previously unpublished domain-specific scheduling algorithm that addresses load balance and data locality simultaneously, which is motivated by queueing theory.
- A novel software managed cache coherency mechanism that handles the explicit movement of data between main memory and multiple hardware accelerators and ultimately facilitates the use of any scheduling algorithm.
- An implementation of an operation that is a key component for a generalized eigensolver that provides exceptional performance on two vastly different computing platforms using SuperMatrix. We use this operation to demonstrate that the methodology provides the infrastructure to easily prototype and support a wide range of dense matrix computations for which the domain-specific scheduling algorithm and software managed cache coherency mechanism can be applied to all of those operations.

Together these contributions provide further evidence of the extensibility and flexibility of our methodology for exploiting parallelism from dense matrix computations.

The rest of the paper is organized as follows. We describe a motivating example used throughout the paper in Section 2. We provide an overview of the basic methodology for exploiting parallelism from dense matrix computations in Section 3. Section 4 proposes a new domain-specific scheduling algorithm. We apply this methodology for use with multiple hardware accelerators in Section 5. Performance results are provided in Section 6. We discuss related work in Section 7 and future work in Section 8. We conclude the paper in Section 9.

2 Reduction from a Symmetric Definite Generalized Eigenproblem to Standard Form

Let $A, B \in \mathbb{R}^{n \times n}$ where A is a symmetric matrix and B is a a symmetric positive definite matrix. One formulation of the generalized eigenproblem is given by

$$ABx = \lambda x$$

where λ is an eigenvalue of AB corresponding to the eigenvector x. Since B is a symmetric positive definite matrix, a lower triangular matrix L can be computed via the Cholesky factorization where

$$B = LL^T$$

so that

$$ALL^T x = \lambda x.$$

By multiplying this equation by L^T,

$$L^T ALL^T x = \lambda L^T x$$

so that

$$My = \lambda y$$

which is a standard eigenproblem where $M = L^T AL$ and $y = L^T x$. As a result, the reduction from a symmetric definite generalized eigenproblem to standard form

$$A \leftarrow L^T AL$$

is computed in order to leverage that A and B are both symmetric matrices instead of explicitly multiplying AB, which destroys symmetry and is computationally expensive.

2.1 Algorithm-by-Blocks

In Figure 1 (left), we present an unblocked algorithm for computing the reduction from a symmetric definite generalized eigenproblem to standard form where α and λ denote scalar elements of A and L, respectively. Typically, this unblocked algorithm is implemented using level-1 and level-2 BLAS [25, 33] routines such as $y := \alpha x + y$ (AXPY), $x := \alpha x$ (SCAL), triangular matrix-vector multiplication (TRMV), and symmetric rank-2 update (SYR2).

In Figure 1 (right), we partition both matrices using a block size b, which results in a matrix of blocks. By viewing submatrix blocks as the fundamental unit of data and operations on blocks, or tasks, as the fundamental unit of

computation, we formulate an *algorithm-by-blocks*. We juxtapose the unblocked algorithm and algorithm-by-blocks in Figure 1 to show that these algorithms are nearly identical except here the operations being performed are now level-3 BLAS [24] routines on submatrix blocks such as general matrix-matrix multiplication (GEMM), triangular matrix-matrix multiplication (TRMM), symmetric matrix-matrix multiplication (SYMM), and symmetric rank-2k update (SYR2K). A recursive subproblem (SYGST) occurs within the algorithm-by-blocks for which we simply invoke the unblocked algorithm.

Each matrix of blocks is typically stored as a hierarchical matrix using one level of blocking in order to gain spatial locality when accessing each block for the execution of a task [26]. Note that the loops in Figure 1 are shown in order to make this paper self-contained. The code that generates the DAG is actually implemented using a stylized application programming interface that encapsulates all the details of storing the tasks and indexing a hierarchical matrix [14, 18, 34].

3 SuperMatrix Runtime System

Instead of exposing the details of parallelization within the code that implements a linear algebra operation, we developed the SuperMatrix runtime system through a clear separation of concerns [18, 20, 21, 39] where we divide the process of exploiting parallelism into two distinct phases: *analyzer* and *dispatcher*. During the analyzer phase, the execution of tasks is delayed, and instead the DAG is constructed dynamically for which only the input and output matrix operands of each task are needed to perform the dependence analysis. Once the analyzer is done, the dispatcher phase is invoked which dispatches and schedules tasks to threads.

3.1 Analyzer

The process for constructing a DAG from an algorithm-by-blocks is quite simple. Instead of executing the tasks as they occur sequentially within the algorithm-by-blocks, the analyzer saves each task onto a global storage space and detects data dependencies between all tasks automatically. Once all tasks have been saved, a DAG is implicitly formed where the tasks represent the vertices of the graph and data dependencies represent the edges.

We present the DAG for the reduction from a symmetric definite generalized eigenproblem to standard form given a 3×3 matrix of blocks in Figure 2. Notice that SYGST_0 overwrites the block $A_{0,0}$ while SYR2K_3 reads and later overwrites that same block. This situation leads to a read-after-write, or flow, data dependency between these two tasks and hence the directed edge in the DAG. Also notice that SYGST_0 occurs from the first iteration of the outer j-loop in Figure 1 (right) while SYR2K_3 occurs from the second iteration of that loop, which represents a loop-carried, inter-iterational data dependency. We can view constructing the DAG as dynamically unrolling the triply nested loops from an algorithm-by-blocks.

The DAG for a 3×3 matrix of blocks is quite trivial, so we also present the DAG given a 10×10 matrix of blocks in Figure 3. As we can see, there exists many opportunities for exploiting parallelism within this large DAG.

Constructing the DAG can be done in parallel with the dispatcher phase, but for this problem domain, the analyzer costs little both in terms of execution time and memory space.

3.2 Dispatcher

Once the DAG has been fully constructed, the dispatcher phase is invoked where the threads are spawned, tasks are scheduled and dispatched to threads, and finally the threads are joined back together once all tasks have been executed. Figure 4 presents the algorithm that every thread performs in order to dispatch and schedule tasks.

Let us represent a DAG as (V, E) where V is the set of vertices and E is the set of edges. In this context, $t_i, t_j \in V$ are vertices in the graph, and $\langle t_i, t_j \rangle \in E$ is a directed edge from t_i to t_j.

If there is a directed edge from vertex t_i to vertex t_j, then t_i is the **parent** of t_j, and t_j is the **child** of t_i. A **root** is a vertex with no parents whereas a **leaf** is a vertex with no children. We define a **ready** task as one that is either a root or all of its parents have been executed. A task is **available** if it is ready and waiting to be executed. A **dependent** task is the child of a given task. We **update** a dependent task by notifying it that one of its parents has been executed. The manner in which the *enqueue* and *dequeue* routines implement a queue of ready and available tasks determines the scheduling of tasks.

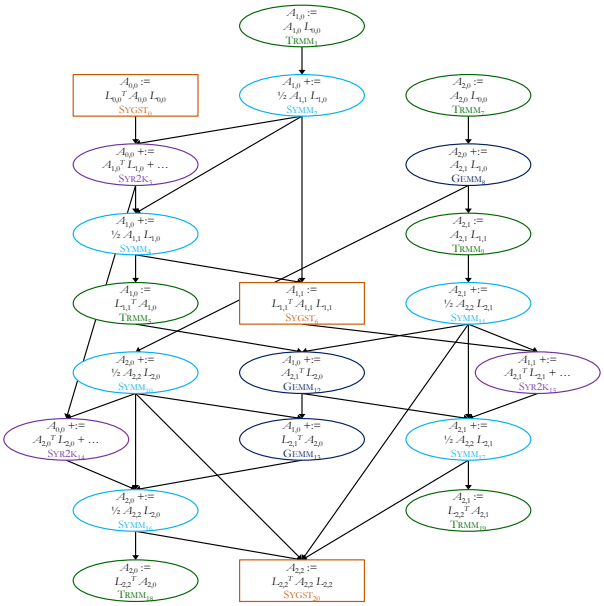

Figure 2: The directed acyclic graph for the reduction from symmetric definite generalized to standard form given a 3×3 matrix of blocks with the input and output operands of each task enumerated. The subscripts within the tasks' names denote the order in which each task occurs sequentially within the algorithm-by-blocks. Here the notation $A \mathrel{+}= B$ represents $A := A + B$, and $A^T B + \ldots$ denotes $A^T B + B^T A$.

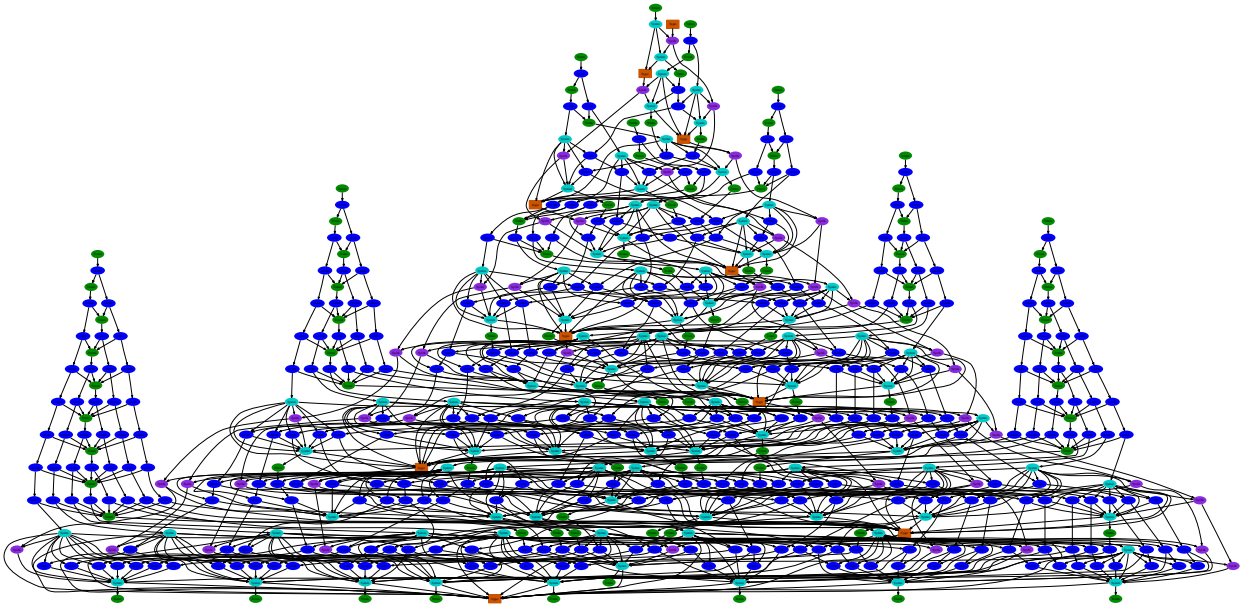

Figure 3: The directed acyclic graph for the reduction from a symmetric definite generalized eigenproblem to standard form given a 10×10 matrix of blocks.

For example, there are three roots in the DAG presented in Figure 2: SYGST_0, TRMM_1, and TRMM_7. All of these roots are enqueued as initial ready tasks by the dispatcher. Some thread will dequeue a task, which is determined by the scheduling algorithm, such as TRMM_7. That thread will execute the task and then update its dependent task GEMM_8, which will then become ready. This process is repeated until all tasks are executed.

The determination of which threads will execute which tasks is done by the enqueue and dequeue routines. Here we discuss two scheduling algorithms for which we motivate using queueing theory: 2D data affinity and a single FIFO queue.

3.2.1 2D Data Affinity

We introduced the idea of data affinity in [20] where tasks are assigned to threads according to a simple owner computes rule. A thread executes all tasks that overwrite a particular block. One task to thread assignment is a two-dimensional block cyclic (2D) distribution where blocks are mapped to a mesh of threads according to its row and column indices within the matrix of blocks. This 2D assignment was inspired by its previous use on distributed-memory architectures in libraries such as ScaLAPACK [15] and PLAPACK [42].

Each thread has its own associated queue from which it dequeues. A thread may need to enqueue ready dependent tasks to the particular queue that task is assigned, which might not be itself.

Data affinity attempts to optimize for data locality. By only allowing one thread to overwrite a particular block, we try to restrict the total number of threads that access a block. If thread affinity is used where a thread is bound to a certain processing element by the operating system, then we reduce data communication through this conceptually simple scheduling algorithm.

3.2.2 Single FIFO Queue

Here all tasks are enqueued at the tail of a single queue, and all threads dequeue tasks from the head, which results in a first-in first-out (FIFO) order. Whenever tasks are ready and available, an idle thread immediately dequeues a task to execute as soon as it gains mutual exclusion to the global queue. Because of this property, this scheduling algorithm reduces the amount of time each thread idles.

3.2.3 Queueing Theory

Queueing theory was first developed to analyze communication networks [27], and it is the study of waiting lines where **customers** from a **source** must wait for a **service** [30]. In this problem domain, the customers are tasks; the source is a DAG; and the servers are threads.

In Figure 5, we depict two multi-server queueing systems. A single-queue multi-server system is on the left where all servers access one queue. A multi-queue multi-server system is on the right where each server has its own dedicated queue.

It is quite intuitive that servicing multiple servers using a single queue is more efficient than doing so from separate queues. There is nonzero probability that a customer will be waiting for a service in one of the p independent queues while another server is idle. On the other hand, that situation cannot occur using a single queue since a customer at the head of the queue is serviced as soon as a server is available. This insight was proven in [40].

2D data affinity implements a multi-queue multi-server system while a single FIFO queue implements a single-queue multi-server system. By reducing the idle time of each thread, a single FIFO queue attempts to maximize load balance, yet no effort is made to address data locality since any task can be executed by any thread. On the other hand, 2D data affinity attempts to minimize data communication by maintaining data locality even though it sacrifices load balance. The issue is how to address both scheduling aspects of load balance and data locality simultaneously.

4 Cache Affinity

We have shown in [20] that 2D data affinity performs particurlarly well because there is a high likelihood that there is a cache hit for the blocks updated by each task. We want to replicate this behavior without the restriction of using multiple queues. In order to do so, we maintain a software cache for each thread that approximates the contents of the physical cache for the associated processing element to which a thread is mapped. The data granularity in this problem domain allows us to amortize the cost of managing these software caches quite efficiently.

```
foreach task in DAG do
    if task is ready then
        Enqueue task
while tasks are available do
    Dequeue task
    Execute task
    foreach dependent task do
        Update dependent task
        if dependent task is ready
        then
            Enqueue dependent
            task
```

Figure 4: The algorithm that all threads execute in order to dispatch and schedule tasks from a directed acyclic graph.

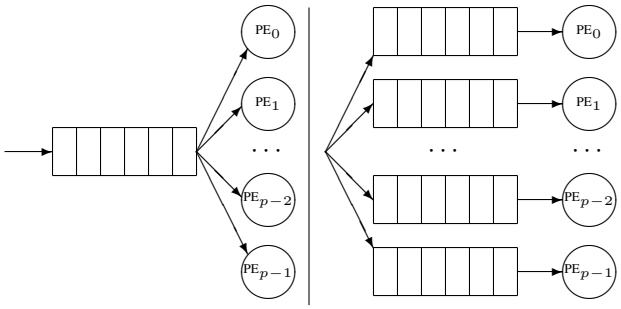

Figure 5: Depictions of a single-queue multi-server system (left) and a multi-queue multi-server system (right), each with p processing elements.

We make several simplifying assumptions. The software cache uses a least recently used (LRU) replacement policy. Each cache line is the size of a submatrix block, and the cache is fully associative. The cache is kept coherent using a write-once protocol.

Our new domain-specific scheduling algorithm with *cache affinity* [18] is a amalgamation of two simple concepts: software caches and a single priority queue.

4.1 Write-Once Cache Coherence Protocol

Here we provide a brief overview of a write-once cache coherence protocol, which is used to provide the shared-memory parallel abstraction [5, 29] where a cache line can be in one of four states. An **invalid** cache line is one that has been invalidated by updates to the same data on another cache, so the data is incoherent. It is **valid** if it has been read into the cache and is not modified. A cache line is **reserved** if it is the only copy of the data in any cache. It is **dirty** if the cache line has been modified, but the data has not been written back to main memory.

When a thread accesses a cache line, only one of four occurrences can happen. On a **read hit**, the data is in the cache, and no state change occurs. A thread reads from main memory for a **read miss**, but if there is a dirty copy in any other cache, it first must be written back to main memory. Any dirty or reserved cache lines will then become valid ones. On a **write hit**, dirty or reserved cache lines are modified and set to dirty. Valid cache lines become reserved, and then the modified data is immediately written back, and all other copies become invalid. A **write miss** invokes the same state changes as a read miss followed by a write hit.

4.2 Priority Queue Heuristics

A key insight in developing this new scheduling algorithm is that all ready and available tasks waiting on a queue can be executed in parallel, so those tasks can be reordered in any fashion. As a result, a priority queue can be used instead of a FIFO one where tasks are sorted according to certain heuristics when they are enqueued.

Incomparable vertices in a DAG are a set of vertices for which there is not a directed **path** between any pair of vertices where a path is a sequence of edges connecting two vertices. As such, incomparable vertices can potentially be executed in parallel. For example in Figure 2, GEMM_{13} and SYMM_{17} are incomparable. When given the choice between scheduling those two tasks, GEMM_{13} could be selected first because it has three **descendants** versus just one for SYMM_{17}. A vertex t_j is a descendant of vertex t_i if there is a directed path from t_i to t_j. This example demonstrates the use of sorting tasks according to the **height** of a vertex where the height is the longest path from a vertex to a leaf.

This heuristic favors tasks on the critical path of execution, which attempts to increase load balance by enabling tasks to become ready and available earlier than they would be if scheduled in FIFO order.

4.3 Overview

The intuition behind cache affinity is quite simple. A single priority queue is used for its load balance properties, and each thread searches that single queue for a task whose operands are already logged on the thread's software cache to preserve data locality.

The dequeue routine now leverages the software cache to provide cache affinity, which is described in Figure 6. By returning a task with a cache hit or the head of the queue, data communication is reduced, or the task with the highest priority is executed. Instead of enqueuing tasks in a FIFO order where priority is set by the arrival times of each task, we modify the enqueue routine to maintain a priority queue by sorting tasks according to their height within the DAG. After dequeuing a task, the dispatcher now must have to update the software caches according to a simplified write-once cache coherence protocol where each cache line is either valid or invalid, ignoring the reserved and dirty states.

When a thread executes a task, the software caches are updated with the following two steps. (1) Update the software cache belonging to the thread that executes the particular task with the task's input and output matrix block operands. If the block already resides in the software cache, there is a cache hit, and the block is marked as most recently used. If there is a cache miss, then the least recently used block is evicted, and the block is logged as most recently used onto the software cache. (2) Write invalidate all copies of the output blocks residing on the other software caches.

The proposed cache affinity scheduling algorithm attempts to address both load balance and data locality simultaneously, which neither 2D data affinity nor a single FIFO queue achieves.

5 Multiple Hardware Accelerators

We have thus far described the basic mechanisms for exploiting parallelism on shared-memory computer architectures where each processing core is a traditional central processing unit (CPU). We now apply this methodology to support the use of multiple hardware accelerators where the key difference is the explicit movement of data between main memory and the memory address space of each accelerator.

5.1 Graphics Processing Units

We primarily handle the case where an accelerator is a graphics processing unit (GPU), yet this methodology can support nearly any type of accelerator. We view a GPU as a single accelerator instead of trying to exploit parallelism within a GPU which contains hundreds of individual streaming processors. We rely on the existence of tuned computational kernels for the execution of tasks on a single GPU, such as CUBLAS [23], which is a BLAS library for GPUs that exploits thread-level parallelism between the many streaming processors.

In order to execute a task on a GPU, the matrix operands of the task must be explicitly moved to the local memory of a GPU. Once a task completes, the updated data must eventually be transferred back to main memory to keep the state of memory consistent.

5.2 Hybrid Execution Model

We assume a very simple model for a heterogeneous platform with multiple hardware accelerators where a CPU spawns threads that are mapped to different GPUs. The CPU in this case may itself also be a homogeneous multicore platform. The threads reside on the CPU and handle the scheduling of tasks. Once a task is selected, the thread dispatches that task to the GPU and handles the data transfers so that a computational kernel can be invoked to execute the task on the GPU.

CUBLAS implements the entire BLAS interface [23, 24, 25, 33], yet very few LAPACK-level kernels are implemented for GPUs. For instance, a GPU kernel for the recursive subproblem of the reduction from a symmetric definite generalized eigenproblem to standard form does not exist at the time of this writing. Instead of painstakingly trying to implement this kernel for GPUs, we simply notice the fact that the CPU can execute an optimized unblocked implementation. As a result, we use a hybrid execution model where both CPU and GPUs perform computation.

When the CPU executes a task, one or more GPUs are left idle. The key to the scalabilty of this approach is that the bulk of the computation lies with tasks that can be executed on GPUs. For the reduction from a symmetric definite generalized eigenproblem to standard form, the only task that is not supported on GPUs is the recursive subproblem, which only occurs on each diagonal block within the algorithm-by-blocks. In Figures 2 and 3, all tasks within an oval as ones that can be executed on a GPU, and the ones within rectangles that must be executed on a CPU. Note that particularly in the larger DAG, only a small fraction of tasks must be executed on a CPU for this operation.

5.3 Software Managed Cache Coherency

No hardware support exists for keeping the memory between multiple GPUs coherent, so we need a mechanism for keeping track of where submatrix blocks are valid or dirty. The advantage of developing the cache affinity scheduling algorithm is that we already have the software cache mechanism for managing the locations of blocks.

The key insight when incorporating multiple GPUs is that the state of the main memory does not need to be consistent until all tasks from a DAG are executed. As a result, we allow dirty blocks to reside on the local memory of a GPU without writing it back to main memory until it is necessary. In doing so, we attempt to minimize the number of data transfers incurred, which are quite costly. This strategy is fundamentally different from our model on CPUs where we assume that updates to blocks are instantly written back to main memory. We make this assumption on CPUs since the hardware implements a cache coherency protocol to provide a consistent shared-memory abstraction.

Instead, we implement a write-back cache coherence protocol for utilizing multiple hardware accelerators, similar to a MSI write-invalidate protocol. The main difference between the write-once and write-back protocols is that the modified data is not written back to main memory on a write hit for the write-back protocol.

In Figure 7, we present the algorithm for dispatching tasks to threads with the protocol for data transfers incorporated. Here we assume that each thread is mapped to an accelerator and that the cache corresponds to the local

memory on that accelerator. The dispatcher is designed with the assumption that different threads cannot access the blocks cached on another accelerator in any way until the data is written back to main memory.

There are several key details to point out within the dispatcher shown in Figure 7. Each thread checks its own cache to see if any blocks have been requested by other threads and subsequently writes those blocks back to main memory before selecting a task. Once a task is selected, all of the task's matrix operands are checked to see whether a block is dirty on another cache and requests that block if so. If all of the task's blocks are valid, then each matrix operand is brought into the cache, and the task is then executed on the accelerator. If any one of the blocks has to be requested on another thread, the dispatcher enqueues the task instead of stalling in order to find another task that might be ready to execute. Once all the tasks in the DAG have been executed, all threads write back every block that is dirty in their respective cache to ensure a consistent state of memory by the end of dispatcher.

In order to support the hybrid execution model, we need to introduce a few modifications to the algorithm in Figure 7. If a task must be executed on a CPU, then the task's blocks that are dirty on the accelerator mapped to that thread must be written back to main memory. Once the task completes execution, all of its output matrix operands must then be write invalidated on the thread's accelerator.

A software managed cache coherence protocol was implemented in [36] that was dependent upon the use of 2D data affinity. Since the mapping of computation to a DAG prescribes the use of an p-reader 1-writer scheme, each block would only be dirty on a single accelerator according to the static 2D distribution. In this paper, we decouple the selection of a task from the data transfers where a block can be dirty in any cache. In doing so, any scheduling algorithm can now be used, e.g. cache affinity, to utilize multiple hardware accelerators.

6 Performance

We compare the performance of the reduction from a symmetric definite generalized eigenproblem to standard form within the SuperMatrix runtime system using different scheduling algorithms on a homogeneous multicore platform and a heterogeneous platform with multiple hardware accelerators.

6.1 Target Architectures

The CPU experiments were performed on a four socket 2.66 GHz Intel Dunnington system running Red Hat 4.1.2-46. This cache coherent non-uniform memory access (ccNUMA) architecture provides a total of 24 processing cores with a theoretical peak performance of 255.36 GFLOPS and 96 GB of general-purpose physical memory. Each socket contains a shared 16 MB L3 cache and three 3 MB L2 caches shared between pairs of cores. The OpenMP implementation provided by the Intel compiler 11.1 served as the underlying threading mechanism used by SuperMatrix. Performance was measured by linking to Intel Math Kernel Library (MKL) 10.2 as a high-performance implementation of the BLAS and LAPACK.

The GPU experiments were performed on a two socket 2.83 GHz Intel Harpertown system running CentOS 5.4, which is connected to an NVIDIA Tesla S1070. The S1070 has four 602 MHz Tesla C1060 GPUs each with 4 GB of DDR3 memory. Each pair of GPUs share a common PCIExpress Gen2 16x slot. The theoretical peak performance of each Tesla C1060 is 933.12 GFLOPS when operating in single precision and 77.76 GFLOPS in double precision. Compute Unified Device Architecture (CUDA) and CUBLAS 2.3 were used in our GPU experiments while the rest of the software is identical to that used on the aforementioned CPU target architecture.

6.2 Implementations

We report the performance in GFLOPS (one billion floating point operations per second) for several implementations of the reduction from a symmetric definite generalized eigenproblem to standard form using single and double precision floating point real arithmetic. We used an operation count of n^3 of useful computation for each implementation presented to calculate the rate of execution. We tuned the storage and algorithmic block size for each problem size when possible. We mapped 24 threads to each of the 24 processing cores on the homogeneous platform and used 4 threads corresponding to each of the 4 GPUs on the heterogeneous platform. We compare three different parallel implementations:

- **SuperMatrix + serial MKL/CUBLAS**
  We use the SuperMatrix runtime system embedded within the open source dense linear algebra library `libflame` [43],

```
foreach task in queue do
    foreach output block of task do
        if block resides in software cache
        then
            Record first task with cache hit
if task has cache hit then
    Return task
else
    Return head task of queue
```

Figure 6: The routine for dequeuing a task when using cache affinity.

```
foreach task in DAG do
    if task is ready then
        Enqueue task
while tasks are available do
    foreach block in cache do
        if block is requested then
            Flush block
            Mark block as valid
    Dequeue task
    foreach block accessed by task do
        foreach thread not itself do
            if block is dirty in thread's cache
            then
                Request block
    if any block was requested then
        Enqueue task
    else
        foreach block accessed by task do
            if block is not in cache then
                if cache is full then
                    Evict LRU block
                Transfer block to cache
            if block is output operand then
                Mark block as dirty
                foreach thread not itself do
                    if block is in thread's cache
                    then
                        Write invalidate block
        Execute task
        foreach dependent task do
            Update dependent task
            if dependent task is ready then
                Enqueue dependent task
foreach block in cache do
    if block is dirty then
        Flush block
```

Figure 7: The algorithm that all threads execute in order to dispatch and schedule tasks from a directed acyclic graph when using multiple hardware accelerators.

which has integrated support for both CPUs and GPUs. We present performance results using the three presented scheduling algorithms: 2D data affinity, a single FIFO queue, and cache affinity. SuperMatrix assumes the use of hierarchical matrices where the submatrix blocks are stored contiguously. We call serial MKL for the execution of tasks on a single processing core and CUBLAS for execution on a GPU.

- **Sequential + Multithreaded MKL/CUBLAS**
  We link a sequential blocked algorithm found in the public domain LAPACK library to MKL multithreaded BLAS routines and CUBLAS on a single GPU, which assumes the matrices are stored in the traditional column-major order storage.

- **Multithreaded MKL**
  On CPUs, we link to MKL multithreaded implementation of `[s,d]sygst`. This implementation exploits parallelism internally and also assumes column-major order storage.

To the best of our knowledge, we provide the only implementation of the reduction from a symmetric definite generalized eigenproblem to standard form using multiple GPUs.

6.3 Results

Performance results are reported in Figure 8. Several comments are in order:

- The performance signatures are nearly the same on CPUs and GPUs where cache affinity is the best performing scheduling algorithm. These results empirically verify that addressing load balance and data locality simultaneously is advantageous versus addressing each separately.

- Linking a sequential algorithm to multithreaded BLAS routines does not provide a scalable solution because inherent synchronization points exist between successive calls to these routines. Despite this limitation, this sequential algorithm still outperforms the multithreaded implementation provided by MKL. This lack of an optimized parallel implementation within a commercial library highlights the need for a systematic methodology for exploiting parallelism such as SuperMatrix.

- Performance when using single and double precision floating point floating point real operations are nearly identical on CPUs because the same floating point units are used on a CPU regardless of the data type being `float` or `double`. On the other hand, GPUs provide a vastly different model where single precision provides much higher performance because of their inherent design being for rendering graphics which only requires single precision. This disparity in performance has narrowed on the next generation NVIDIA Fermi GPU architecture, which provides further hardware support for double precision floating point operations.

- Even though we can achieve over two-thirds of the theoretical peak performance on CPUs, we only utilize a mere fraction of the peak performance on GPUs. Despite this limitation, our use of four GPUs provides much higher absolute performance than 24 processing cores when using single precision.

In Figure 9, we compare the speedup using cache affinity versus the sequential blocked algorithm on both CPUs and GPUs. This scheduling algorithm scales when using more computational threads even when using a prime number of threads for which 2D data affinity does not particularly handle well. Even though we only achieve a fraction of the theoretical peak performance on GPUs, SuperMatrix attains strong scalability.

In Figure 10 (left), we compare the load balance properties of the three different scheduling algorithms presented in this paper. We calculate load balance by summating the total time each thread performs useful computation and dividing that sum by the number of threads and the total parallel execution time of the dispatcher so that a higher ratio equates to better load balance. On CPUs, cache affinity and a single FIFO queue exhibit nearly the same load balance while 2D data affinity exhibits poor load balance in comparison on CPUs as predicted by queueing theory. On GPUs, 2D data affinity has rather high load balance since we only use a small number of computational threads while a single FIFO queue has poor load balance for larger problem sizes because the cost of extra data transfers skews this ratio.

In Figure 10 (right), we compare the data locality properties of the scheduling algorithms. We calculate the simulated cache hit ratio on CPUs by summating all the tasks for which its output matrix operand records a cache hit within the software cache of a thread and dividing that sum by the total number of tasks. We calculate the data transfer rate on GPUs by summating all the occurrences when a data transfer is not needed before or after executing a task

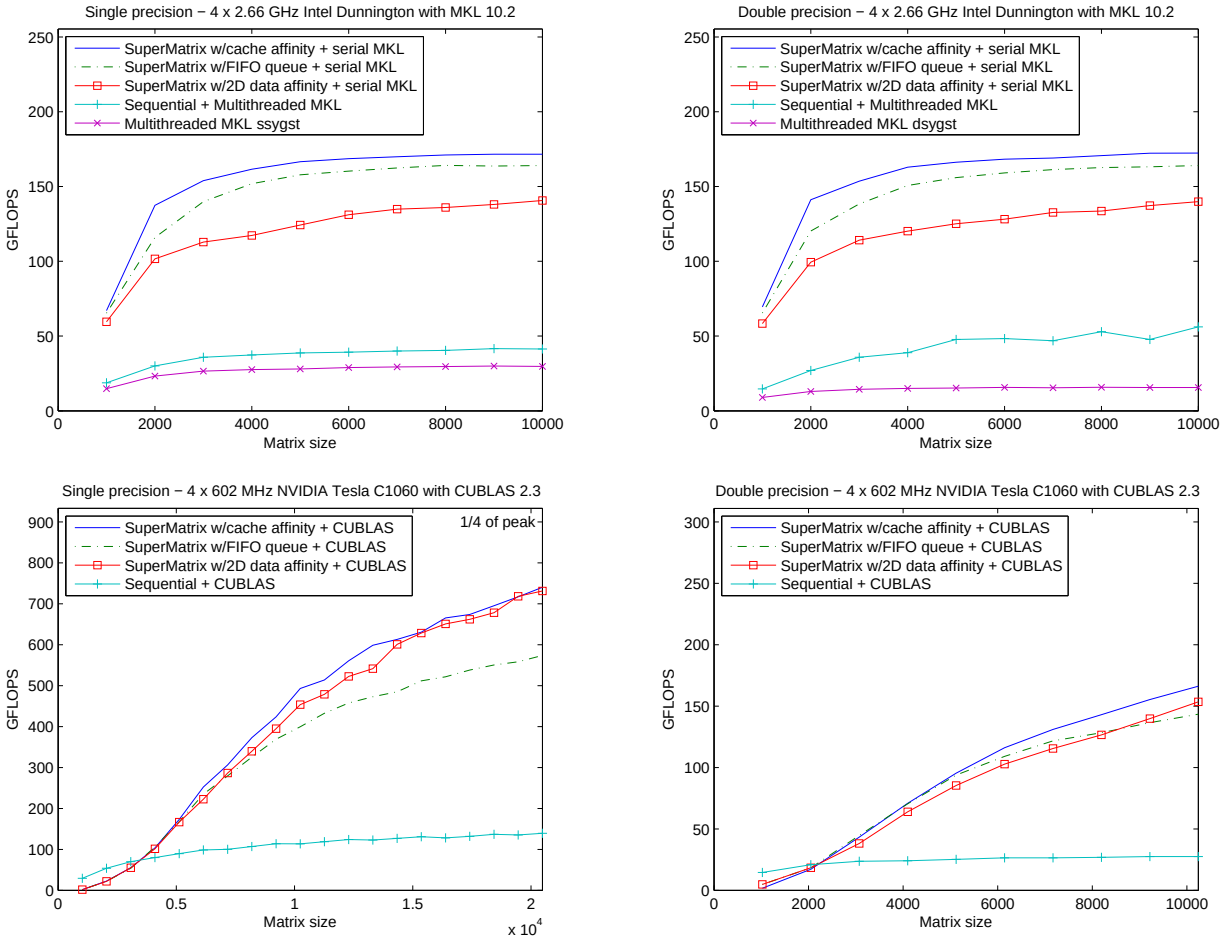

Figure 8: Performance of different implementations for the reduction from a symmetric definite generalized eigenproblem to standard form on a homogeneous CPU platform (top) and a heterogeneous platform with multiple GPUs (bottom) using single (left) and double (right) precision floating point real arithmetic.

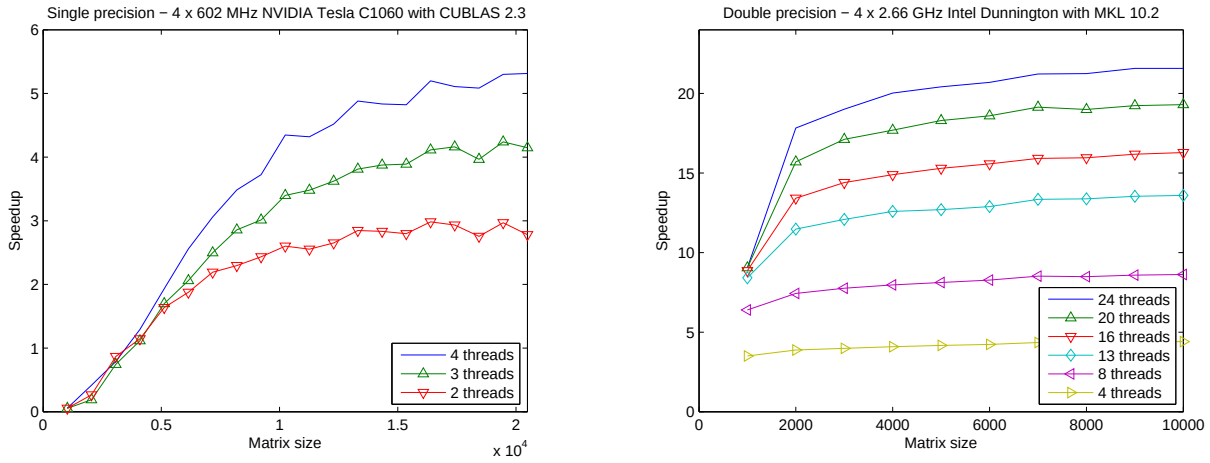

Figure 9: Speedup of SuperMatrix using cache affinity versus a sequential blocked algorithm linked to serial MKL/CUBLAS for the reduction from a symmetric definite generalized eigenproblem to standard form on a heterogeneous platform with multiple GPUs using single precision (left) and a homogeneous CPU platform using double precision (right).

and dividing that sum by twice the number of blocks accessed by every task since each block theoretically needs to be moved to and from the GPU. A higher ratio for both of these measures equates to reducing data communication and thus preserving data locality. Even though cache affinity and a single FIFO queue have the same load balance, the varying data locality accounts for the difference in their performance curves shown in Figure 8. As the problem size grows, the likelihood that a block resides in the cache diminishes when using 2D data affinity on CPUs. This situation occurs because the cache size is fixed and rather small compared to GPUs while the number of blocks assigned to a particular thread grows with the problem size.

7 Related Work

We only compare three pertinent scheduling algorithms in this paper. For a more thorough examination of scheduling algorithms for this problem domain including work stealing, see [18]. We have classical work stealing [16] and the mailbox optimization for work stealing that addresses data locality [1] implemented within the SuperMatrix runtime system embedded within the `libflame` library [43], both of which can be used on CPUs and GPUs.

The related projects Parallel Linear Algebra for Scalable Multi-core Architectures (PLASMA) [3, 17] and Matrix Algebra on GPU and Multicore Architectures (MAGMA) [35, 41] apply similar DAG scheduling principles for parallelizing dense matrix computations for CPUs and GPUs, respectively. Neither of these projects supports the reduction from a symmetric definite generalized eigenproblem to standard form for the moment. These projects have only started investigating dynamic scheduling algorithms, partially for inversion of a symmetric positive definite matrix [2], for which we have studied using a single FIFO queue and 2D data affinity several years ago [20, 21].

Many runtime systems exist that mapping computation to a DAG using the input and output parameters of a task such as Cell Superscalar (CellSs) [9, 10, 11], the Star Superscalar (StarSs) programming model [8], and StarPU [6, 7]. These runtime systems perform the dynamic scheduling of tasks for heterogeneous computing platforms but are not domain-specific to dense matrix computations.

A hybrid execution model was also used to program dense matrix computations in [44], yet their methodology was only designed for a single GPU whereby exploiting parallelism between the streaming processors within a GPU.

8 Future Work

We have three specific items that we want to address in future work:

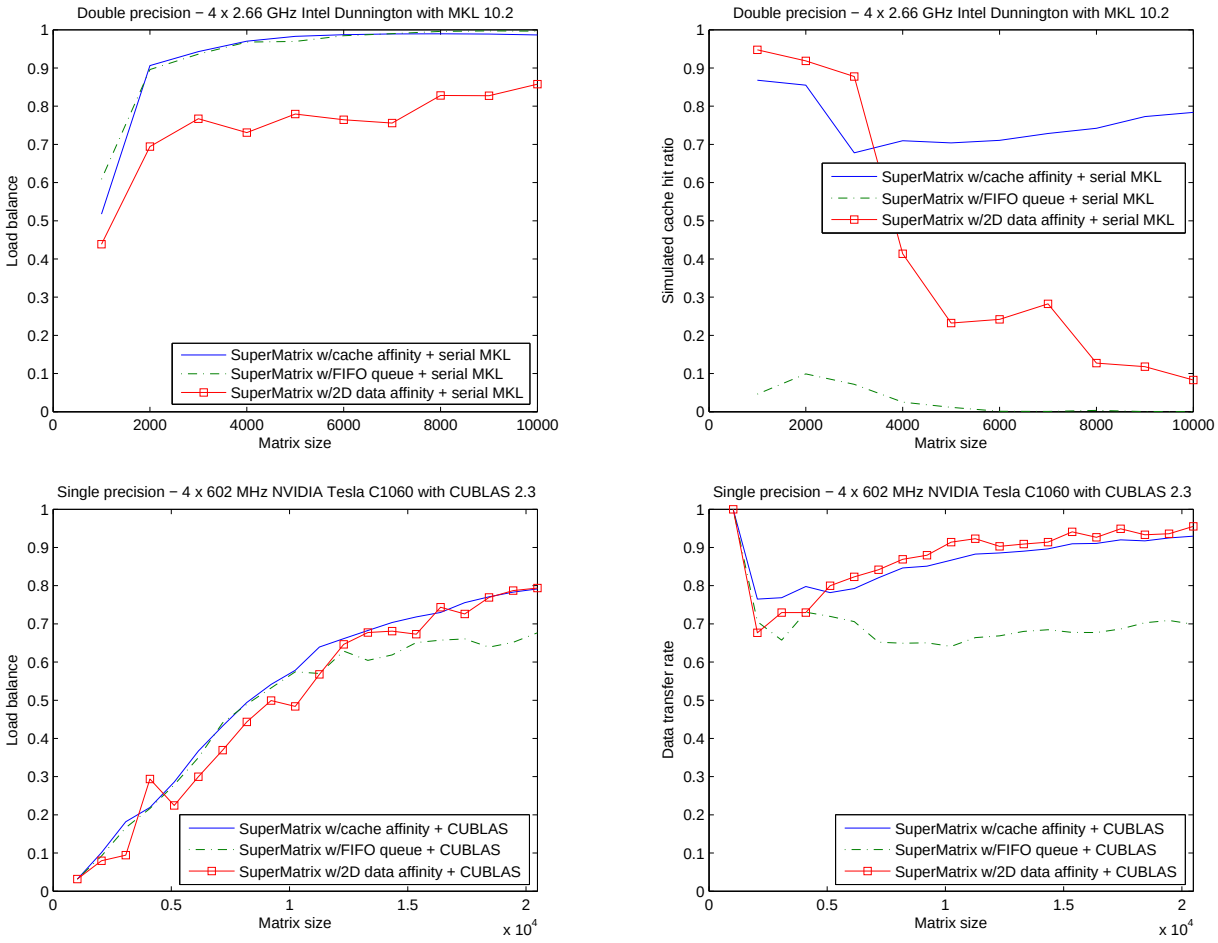

Figure 10: Load balance (left) and data locality (right) properties of different scheduling algorithms within SuperMatrix for the reduction from a symmetric definite generalized eigenproblem to standard form on a homogeneous CPU platform using double precision (top) and a heterogeneous platform with multiple GPUs using single precision (bottom).

- The heterogeneous platform with multiple GPUs contains eight processing cores, yet we only spawn four threads that correspond to each of the four GPUs. As a result, four CPUs are being left idle. We can extend the runtime system to utilize these extra processing cores and prioritize tasks that can only be executed on CPUs to these extra processing cores.
- Within our software managed cache coherency mechanism, we implement a fully associative software cache. When checking if a block is dirty in another thread's cache, all blocks within the cache must be evaluated. Instead, we can implement a set associative cache to reduce the number of blocks that must be searched before executing a task.
- Several kernels within CUBLAS are not well optimized compared to GEMM [32] such as TRMM, SYMM, and SYR2K. We can reimplement some of these kernels for GPUs to attain higher performance when executing tasks that invoke these kernels.

By tackling these issues, we believe that we can further increase performance when using multiple hardware accelerators.

9 Conclusion

In this paper, we describe a systematic methodology for parallelizing dense matrix computations that utilizes both homogeneous multicore platforms and heterogeneous platforms with multiple hardware accelerators. We develop the cache affinity scheduling algorithm that addresses load balance and data locality simultaneously, which is motivated by queueing theory and subsequently provides superior performance on both computing platforms. It is the separation of concerns implemented by the SuperMatrix runtime system that allows us to seamlessly use different scheduling algorithms and support the data transfers necessary to utilize multiple hardware accelerators.

We conclude with an anecdote about the productivity afforded by using our methodology. We rederived a blocked algorithm for the reduction from a symmetric definite generalized eigenproblem to standard form found in the public domain LAPACK library and reformulated it as an algorithm-by-blocks in about half an hour using the FLAME methodology [12, 13, 14, 31, 34]. Afterwards, we implemented this operation within the `libflame` library [43] with full support for error checking, test routines, and all four floating point data types, including complex arithmetic, along with linking to SuperMatrix for both CPUs and GPUs in just four hours!

Additional Information

For additional information on the Formal Linear Algebra Methods Environment (FLAME) visit
`http://www.cs.utexas.edu/users/flame/`.

Acknowledgements

We thank the other members of the FLAME team for their support, namely Paolo Bientinesi, Diego Fabregat-Traver, Enrique S. Quintana-Ortí, Gregorio Quintana-Ortí, Robert van de Geijn, and Field G. Van Zee. We also thank NVIDIA for the generous donation of part of the graphics hardware used in the experiments.

This research is partially sponsored by Intel and Microsoft Corporations and NSF grants CCF–0540926 and CCF–0702714. *Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation (NSF).* The research at the Universidad Jaume I is supported by the Spanish Ministry of Science and Innovation/FEDER (contract no. TIN2008-06570-C04-01).

References

[1] U. A. Acar, G. E. Blelloch, and R. D. Blumofe. The data locality of work stealing. In *SPAA '00: Proceedings of the Twelfth ACM Symposium on Parallel Algorithms and Architectures*, pages 1–12, Bar Harbor, ME, USA, July 2000.

[2] E. Agullo, H. Bouwmeester, J. Dongarra, J. Kurzak, J. Langou, and L. Rosenberg. Towards an efficient tile matrix inversion of symmetric positive definite matrices on multicore architectures. In *VecPar '10: Proceedings of the Ninth International Meeting on High Performance Computing for Computational Science*, Berkeley, CA, USA, June 2010.

[3] E. Agullo, B. Hadri, H. Ltaief, and J. Dongarra. Comparative study of one-sided factorizations with multiple software packages on multi-core hardware. In *SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pages 1–12, Portland, OR, USA, November 2009.

[4] E. Anderson, Z. Bai, J. Demmel, J. E. Dongarra, J. DuCroz, A. Greenbaum, S. Hammarling, A. E. McKenney, S. Ostrouchov, and D. Sorensen. *LAPACK Users' Guide*. SIAM, Philadelphia, 1992.

[5] J. Archibald and J.-L. Baer. Cache coherence protocols: Evaluation using a multiprocessor simulation model. *ACM Transactions on Computer Systems*, 4(4):273–298, November 1986.

[6] C. Augonnet, S. Thibault, and R. Namyst. Automatic calibration of performance models on heterogeneous multicore architectures. In *Euro-Par '09: Proceedings of the Fifteenth International Euro-Par Conference on Parallel Processing*, pages 56–65, Delft, the Netherlands, August 2009.

[7] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier. StarPU: A unified platform for task scheduling on heterogeneous multicore architectures. In *Euro-Par '09: Proceedings of the Fifteenth International Euro-Par Conference on Parallel Processing*, pages 863–874, Delft, the Netherlands, August 2009.

[8] E. Ayguadé, R. M. Badia, F. D. Igual, J. Labarta, R. Mayo, and E. S. Quintana-Ortí. An extension of the StarSs programming model for platforms with multiple GPUs. In *Euro-Par '09: Proceedings of the Fifteenth International Euro-Par Conference on Parallel Processing*, pages 851–862, Delft, the Netherlands, August 2009.

[9] P. Bellens, J. M. Pérez, R. M. Badia, and J. Labarta. Exploiting locality on the Cell/B.E. through bypassing. In *SAMOS '09: Proceedings of the Ninth International Workshop on Embedded Computer Systems: Architectures, Modeling, and Simulation*, pages 318–328, Samos, Greece, July 2009.

[10] P. Bellens, J. M. Pérez, R. M. Badia, and J. Labarta. Just-in-time renaming and lazy write-back on the Cell/B.E. In *ICPPW '09: Proceedings of the 2009 International Conference on Parallel Processing Workshops*, pages 138–145, Vienna, Austria, September 2009.

[11] P. Bellens, J. M. Pérez, F. Cabarcas, A. Ramírez, R. M. Badia, and J. Labarta. CellSs: Scheduling techniques to better exploit memory hierarchy. *Scientific Programming*, 17(1-2):77–95, January 2009.

[12] P. Bientinesi. *Mechanical Derivation and Systematic Analysis of Correct Linear Algebra Algorithms*. PhD thesis, The University of Texas at Austin, 2006.

[13] P. Bientinesi, J. A. Gunnels, M. E. Myers, E. S. Quintana-Ortí, and R. A. van de Geijn. The science of deriving dense linear algebra algorithms. *ACM Transactions on Mathematical Software*, 31(1):1–26, March 2005.

[14] P. Bientinesi, E. S. Quintana-Ortí, and R. A. van de Geijn. Representing linear algebra algorithms in code: The FLAME application programming interfaces. *ACM Transactions on Mathematical Software*, 31(1):27–59, March 2005.

[15] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. *ScaLAPACK Users' Guide*. SIAM, Philadelphia, 1997.

[16] R. Blumofe and C. Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM*, 46(5):720–748, September 1999.

[17] A. Buttari, J. Langou, J. Kurzak, and J. Dongarra. A class of parallel tiled linear algebra algorithms for multicore architectures. *Parallel Computing*, 35(1):38–53, January 2009.

[18] E. Chan. *Application of Dependence Analysis and Runtime Data Flow Graph Scheduling to Matrix Computations*. PhD thesis, The University of Texas at Austin, 2010.

[19] E. Chan, A. Chapman, and R. van de Geijn. Managing the complexity of lookahead for LU factorization with pivoting. In *SPAA '10: Proceedings of the 22nd ACM Symposium on Parallelism in Algorithms and Architectures*, pages 200–208, Santorini, Greece, June 2010.

[20] E. Chan, E. S. Quintana-Ortí, G. Quintana-Ortí, and R. van de Geijn. SuperMatrix out-of-order scheduling of matrix operations for SMP and multi-core architectures. In *SPAA '07: Proceedings of the Nineteenth ACM Symposium on Parallelism in Algorithms and Architectures*, pages 116–125, San Diego, CA, USA, June 2007.

[21] E. Chan, F. G. Van Zee, P. Bientinesi, E. S. Quintana-Ortí, G. Quintana-Ortí, and R. van de Geijn. SuperMatrix: A multithreaded runtime scheduling system for algorithms-by-blocks. In *PPoPP '08: Proceedings of the Thirteenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 123–132, Salt Lake City, UT, USA, February 2008.

[22] E. Chan, F. G. Van Zee, E. S. Quintana-Ortí, G. Quintana-Ortí, and R. van de Geijn. Satisfying your dependencies with SuperMatrix. In *Cluster '07: Proceedings of the 2007 IEEE International Conference on Cluster Computing*, pages 91–99, Austin, TX, USA, September 2007.

[23] *CUBLAS User Guide*. `http://developer.nvidia.com`, 2010.

[24] J. J. Dongarra, J. Du Croz, S. Hammarling, and I. Duff. A set of level 3 basic linear algebra subprograms. *ACM Transactions on Mathematical Software*, 16(1):1–17, March 1990.

[25] J. J. Dongarra, J. Du Croz, S. Hammarling, and R. J. Hanson. An extended set of Fortran basic linear algebra subprograms. *ACM Transactions on Mathematical Software*, 14(1):1–17, March 1988.

[26] E. Elmroth, F. Gustavson, I. Jonsson, and B. Kagstrom. Recursive blocked algorithms and hybrid data structures for dense matrix library software. *SIAM Review*, 46(1):3–45, 2004.

[27] A. K. Erlang. The theory of probabilities and telephone conversations. *Nyt Tidsskrift for Matematik*, 20(B):33–39, 1909.

[28] D. Geer. Industry trends: Chip makers turn to multicore processors. *Computer*, 38(5):11–13, May 2005.

[29] J. R. Goodman. Using cache memory to reduce processor-memory traffic. In *ISCA '83: Proceedings of the 10th Annual International Symposium on Computer Architecture*, pages 124–131, Stockholm, Sweden, 1983.

[30] D. Gross and C. M. Harris. *Fundamentals of Queueing Theory (3rd ed.)*. John Wiley & Sons, New York, 1998.

[31] J. A. Gunnels, F. G. Gustavson, G. M. Henry, and R. A. van de Geijn. FLAME: Formal linear algebra methods environment. *ACM Transactions on Mathematical Software*, 27(4):422–455, December 2001.

[32] F. D. Igual, G. Quintana-Ortí, and R. van de Geijn. Level-3 BLAS on a GPU: Picking the low hanging fruit. In *ICNAAM '09: Proceedings of the Seventh International Conference of Numerical Analysis and Applied Mathematics*, Crete, Greece, September 2009.

[33] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic linear algebra subprograms for Fortran usage. *ACM Transactions on Mathematical Software*, 5(3):308–323, September 1979.

[34] T. M. Low and R. van de Geijn. An API for manipulating matrices stored by blocks. FLAME Working Note #12 TR-04-15, The University of Texas at Austin, Department of Computer Sciences, May 2004.

[35] H. Ltaief, S. Tomov, R. Nath, and J. Dongarra. Hybrid multicore Cholesky factorization with multiple GPU accelerators. *IEEE Transaction on Parallel and Distributed Systems*, 2010. Submitted.

[36] G. Quintana-Ortí, F. D. Igual, E. S. Quintana-Ortí, and R. A. van de Geijn. Solving dense linear systems on platforms with multiple hardware accelerators. In *PPoPP '09: Proceedings of the Fourteenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 121–130, Raleigh, NC, USA, February 2009.

[37] G. Quintana-Ortí, E. S. Quintana-Ortí, E. Chan, R. van de Geijn, and F. G. Van Zee. Design and scheduling of an algorithm-by-blocks for LU factorization on multithreaded architectures. In *MTAAP '08: Workshop on Multithreaded Architectures and Applications*, Miami, FL, USA, April 2008.

[38] G. Quintana-Ortí, E. S. Quintana-Ortí, E. Chan, R. A. van de Geijn, and F. G. Van Zee. Scheduling of QR factorization algorithms on SMP and multi-core architectures. In *PDP '08: Sixteenth Euromicro International Conference on Parallel, Distributed and Network-Based Processing*, pages 301-310, Toulouse, France, February 2008.

[39] G. Quintana-Ortí, E. S. Quintana-Ortí, R. A. van de Geijn, F. G. Van Zee, and E. Chan. Programming matrix algorithms-by-blocks for thread-level parallelism. *ACM Transactions on Mathematical Software*, 36(3):14:1-14:26, July 2009.

[40] D. R. Smith and W. Whitt. Resource sharing for efficiency in traffic systems. *Bell System Technical Journal*, 60(1):39–55, January 1981.

[41] S. Tomov, R. Nath, H. Ltaief, and J. Dongarra. Dense linear algebra solvers for multicore with GPU accelerators. In *HIPS '10: Proceedings of the Fifteenth International Workshop on High-Level Parallel Programming Models and Supportive Environments*, Atlanta, GA, USA, April 2010.

[42] R. A. van de Geijn. *Using PLAPACK: Parallel Linear Algebra Package*. MIT Press, Cambridge, 1997.

[43] F. G. Van Zee. libflame*: The Complete Reference*. http://www.lulu.com, 2009.

[44] V. Volkov and J. W. Demmel. Benchmarking GPUs to tune dense linear algebra. In *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–11, Austin, TX, USA, November 2008.