Task-Based Scheduling Implementing Tiled Linear Algebra on GPUs

Abstract

Context/Background

As microprocessors have moved away from increasing clock speed to increasing the number of cores on a microchip in persuit of Moore's law, algorithms are being reformulated at a basic level to make use of the parallel processing power in modern computing platforms. The algorithm explored here is a reorganisation of a common linear algebra operation, the QR factorisation, using task-based scheduling. GPUs are exemplary of this paradigm shift, as they are the most stark example of the move to massively parallel architectures in scientific computing, and NVidia's CUDA has proven to be a useful tool in scientific computing, opening up massively parallel computing to a wide academic audience.

Aims

The aim for the project will be to explore using task-based parallelism to implement a QR decomposition algorithm designed for fine-grain parallelism on the GPU with CUDA homogeneously. The aim is to investigate the scalability and running time of the algorithm when implemented with task-based scheduling compared to both a naiive CPU implementation and other implementations using the GPU as an accelerator in a hetrogeneous system.

Method

Task-based parallelism on the GPU with CUDA to compute a tiled QR decomposition will be implemented. Timing, resource use and performance data will be collected and compared to that collected from runs of other GPU implementations of the same operation using the GPU only as an accelerator. Different types of matrix will be investigated, ranging from Tall and Skinny (TS) to square. Conclusions will be drawn as to the viability of further development of tiled linear algebra operations using task-based scheduling on CUDA.

Proposed Solution

The proposed solution will include primarily a GPU task-based parallel scheduler and secondarily an implementation of tiled QR for GPU, using task-based scheduling. The solution will apply a published algorithm known to work well in a highly parallel CPU environment however we apply a different implementation of task-based scheduling to that of the authors.

Keywords: task-based scheduling, cuda, qr decomposition, gpgpu.

I INTRODUCTION

A *Task-based Parallelism*

Task-based parallelism is a model of parallel computation where, unlike data parallelism, the computation is broken down into different tasks which depend on and scale with the same data, as opposed to the same tasks applied to different data. A more detailed description of the Cilk runtime system, which exemplifies task-based scheduling is (Blumofe et al. 1995). The pattern consists of modelling working units, threads, as capable of working on whatever is currently available, or ready, to be processed depending on the data. Once processing, the thread locks the data so it can not be processed by other threads at the same time.

Almost always in the execution of an algorithm based around task-based scheduling, there will be some dependency between tasks; data used for a task that has to be written by a different task, for example. The dependencies between tasks are most commonly modelled as a Directed Acyclic Graph, a DAG, and tasks distributed amongst working units and executed concurrently such that the dependencies are not violated.

The task-based approach to parallelism is well suited to fine-grain parallel algorithms, that is algorithms which operate on small portions of data locally, where the overall computation is broken down into tasks performing different procedures on the data. Recent tiled algorithms for linear algebra operations presented in (Buttari et al. 2009), exemplify this model, as typically they operate with on small contiguous portions of a matrix in several different ways, dependant on where the data, or tile, is in the matrix. Tiled algorithms also have dependencies between tasks, enabling a task scheduling strategy to be applied. This project will implement a tiled QR decomposition algorithm with task-based scheduling to run entirely on the GPU using CUDA.

B *Householder QR Decomposition*

The QR Decomposition of a matrix $A \in \mathbb{R}^{m \times n}$ (where $m \geq n$) is a factorisation of A into two matrices $R \in \mathbb{R}^{m \times n}$ and $Q \in \mathbb{R}^{m \times m}$, such that Q is orthogonal ($Q^T Q = I$), R is upper-triangular and $A = QR$ (Golub & van Loan 1989). One method of performing such a factorisation uses householder matrices to zero successive columns of A below the diagonal, storing R in the upper-triangular portion of A. Householder matrices are matrices of the form

$$P = I - 2\frac{vv^T}{v^T v} \tag{1}$$

where v is given by

$$v = x + \text{sign}(x_1)||x||_2 e_1 \tag{2}$$

where x is a column of the matrix to be factorised, starting with the first. For convenience, v is normalised by dividing by its first element, making $v_1 = 1$. Because of the extra computation required to compute Q from the vectors, and that they can be stored in place, Q is rarely formed explicitly; if a product Qx is required, the vectors stored are applied to x sequentially (Trefethen & Bau 1997). The householder algorithm proceeds according to Algorithm 1, where upon completion, A is left in upper triangular form and the essential $v(2:m)$ portions of the householder vectors below the diagonal. The algorithm runs in $2mn^2 - \frac{2}{3}n^3 = O(n^3)$ flops.

The individual steps of this algorithm could be formulated well for a parallel environment. However there are synchronisation points after each line, and at the end of each step k. Thus, it is

Algorithm 1 Householder QR Decomposition

1: **for** $k = 1 \rightarrow n$ **do**
2: $x \leftarrow A(k:m, k)$
3: $v \leftarrow x + \text{sign}(x_1)||x||_2 e_1$
4: $v \leftarrow v/v_1$
5: $A(k:m, k:n) \leftarrow A(k:m, k:n) - 2\frac{vv^T}{v^T v}A(k:m, k:n)$
6: $A(k+1:m, k) \leftarrow v(2:m)$
7: **end for**

a sequential algorithm that could rely on parallel building blocks, thus it does not have the level of fine-grain parallelism required. Regular synchronisation does not allow for efficient parallel implementation of an algorithm, as all threads have to wait before computation can proceed.

C Tiled QR decomposition

(Buttari et al. 2009) proposed a re-formulation of the sequential algorithm to acheive fine granularity parallelism, with the operations "tiled" and operating on small blocks. Tiling the QR factorisation required a change in the QR algorithm to allow for fine grain tasks operating on small tiles of data. This resulted in 4 elementary operations which can be scheduled asynchronously according to the graph of their dependencies. These dependencies are represented in a Directed Acyclic Graph (Buttari et al. 2006) where the nodes represent tasks and arcs from a node t_1 to t_2 reprents that t_1 writes data that t_2 requires, thus t_2 cannot execute until t_1 has finished executing. Scheduling these tasks asynchronously according to their dependency DAG results in a fine grain algorithm for QR decomposition.

The 4 operations for factorising a matrix A with block size $b << n$ are as follows (Buttari et al. 2008):

1. `DGEQT2`. This operation computes the householder QR factorisation of a tile on the diagonal of A, A_{kk}, of size $b \times b$, overwriting the upper-triangular portion with R_{kk} and storing the householder vectors in place below the diagonal. A temporary workspace is needed to store the vector v during the computation.
   Thus `DGEQT2`(A_{kk}) performs
   $$A_{kk} \leftarrow V_k k, R_k k \tag{3}$$

2. `DLARFB`. This operation applies the vectors computed by a factorisation of a diagonal tile, an implicit Q^T, to a tile on the same row.
   Thus `DLARFB`(V_{kk}, A_{kj}) performs
   $$A_{kj} \leftarrow Q_{kk}^T A_{kj} \tag{4}$$

3. `DTSQT2`. This operation performs the householder QR factorisation of a matrix formed by coupling a upper-right triangular diagonal tile Rkk on top of a tile directly below the diagonal, A_{ik}, resulting in a new upper triangular tile $\hat{R}_{kk}$. As the factorisation results in an identity block on top of a block of non-zero householder vectors, there is no need to rewrite the lower-triangular portion of R_{kk} with the top block of vectors, as the identity

block can be implicit.
Thus `DTSQT2`(R_{kk}, A_{ik}) performs

$$\begin{pmatrix} R_{kk} \\ A_{ik} \end{pmatrix} \leftarrow \begin{pmatrix} \hat{R}_{kk} \\ V_{ik} \end{pmatrix} \tag{5}$$

4. `DSSRFB`. This operation applies the vectors produced by `DTSQT2`, an implicit Q^T, taking into account the implicit identity block, to a matrix formed by coupling a diagonal-row tile on top of a tile below in the same column.
Thus `DSSRFB`(V_{ik}, A_{kj}, A_{ij}) performs

$$\begin{pmatrix} A_{kj} \\ Aij \end{pmatrix} \leftarrow Q_{ik}^T \begin{pmatrix} A_{kj} \\ Aij \end{pmatrix} \tag{6}$$

Assuming a matrix of size $pb \times qb$, these operations come together in Algorithm 2:

Algorithm 2 Tiled QR

```
for k = 1 →min(p, q) do
    DGEQT2(A_kk)
    for j = k + 1 → q do
        DLARFB(V_kk, A_kj)
    end for
    for i = k + 1 → p do
        DTSQT2(R_kk, A_ik)
        for j = k + 1 → q do
            DSSRFB(V_ik, A_kj, A_ij)
        end for
    end for
end for
```

D Project Purpose

The purpose of this project is to explore the implementation of task-based scheduling, to apply a published parallel tiled QR algorithm on GPUs with CUDA. We will look to answer the question: "Is task-based parallelism on GPUs a more efficient method of computing established tiled linear algebra operations, specifically for the tiled QR operation, than the current GPU-accelerated implementations?"

E Deliverables

- The basic deliverable is a correct CPU implementation of tiled QR decomposition using task-based scheduling. This serves to work as the foundation for the Intermediate and Advanced deliverables as it will help in understanding the tiled algorithm, but more importantly will develop the mathematical and scheduling logic and data structures which will be used in the later implementations. This deliverable will also be used as a baseline for

Table 1: Functional Requirements for the project

Deliverable	ID	Description	Priority
Basic	FR1	The implementation of a serial CPU tiled QR decomposition using task-based scheduling.	high
Basic	FR2	Improving the serial CPU decomposition to support parallelism, utilising task-based scheduling to schedule tasks amongst some concurrent threads.	high
Intermediate	FR3	Implement a GPU-based CUDA tiled QR decomposition reusing the task-based scheduling logic from FR2 to compute a parallel decomposition directly on the GPU.	high
Advanced	FR4	Implement a method of scheduling general tasks with interdependencies on the GPU.	medium

performance comparisons with the GPU implementations. This deliverable is also crucial because it requires a strong grasp of both the algorithm itself but more importantly the dependencies and proper management of the tasks in the decomposition.

- The intermediate deliverable consists of an implementation of the tiled QR algorithm using task-based scheduling to schedule tasks on the GPU. The majority of the work here will be in translating the operations and logic from the scheduling and task-processing in the basic deliverable into code which will run efficiently in the highly parallel CUDA GPU environment.
- The advanced deliverable is a multi-purpose, configurable scheduling system for task-based parallelism on the GPU. The aim of this project is to explore the possibility of task-based scheduling for linear algebra on GPUs and this deliverable is a system which would enable a developer to simply integrate their tasks in to our scheduler, specify the dependencies and the process would work as expected, homogeneously on the GPU.

II DESIGN

A Functional Requirements

The functional requirements for the project are presented in Table 1.

B Algorithms & Data Structures

The specific details of implementing the four operations divided loosely into two types: factorisations and applications.

Factorisations: For the single factorisation operation, `DGEQT2`, a sequence of householder vector formation, normalisation and application is performed as per Algorithm 1. Lines 2 to 4 are of less interest to us, but line 5 can be organised in two different ways. The first of these is explicitly forming a matrix

$$P = I - 2\frac{vv^T}{v^Tv} \tag{7}$$

then forming the product PA. This is impractical and slow, as an unneccessary matrix-matrix product, an $O(n^3)$ operation, is required, as well as the storage for P. The answer here is to directly compute each element of the result in place. The problem for each element of A, a_{ij} can be reformulated to the form

$$a_{ij} = a_{ij} + y_i z_j \tag{8}$$

where each element of the vector y, y_i,and each element of the single-row matrix z, z_j can be written as

$$y_i = v_i \frac{-2}{\sum_{k=1}^{m} v_k^2}, z_j = \sum_{k=1}^{m} a_{kj} v_k \tag{9}$$

Thus the formula can be rewritten as

$$a_{ij} = a_{ij} + \left(v_i \frac{-2}{\sum_{k=1}^{m} v_k^2} \right) \left(\sum_{k=1}^{m} a_{kj} v_k \right) \tag{10}$$

revealing the $O(n^2)$ Algorithm 3 for an operation previously taking $O(n^3)$ flops.

Algorithm 3 Calculate the update of A

1: $y \leftarrow 0$
2: **for** $k = 1 \rightarrow m$ **do**
3: $\quad y \leftarrow y + v_k^2$
4: **end for**
5: $y \leftarrow -2 \div y$
6: **for** $j = 1 \rightarrow n$ **do**
7: $\quad z \leftarrow 0$
8: $\quad$ **for** $k = 1 \rightarrow m$ **do**
9: $\quad\quad z \leftarrow z + A_{kj} \times v_k$
10: $\quad$ **end for**
11: $\quad$ **for** $i = 1 \rightarrow m$ **do**
12: $\quad\quad t \leftarrow y \times z$
13: $\quad\quad t \leftarrow t \times v_i$
14: $\quad\quad A_{ij} \leftarrow A_{ij} + t$
15: $\quad$ **end for**
16: **end for**

The double factorisation operation, `DTSQT2` is the same sequence of tasks as above, but with an upper-triangular top tile. The top tile contains vectors from previous factorisations below the diagonal, which must be preserved. The algorithm proceeds similarly to the single factorisation case, and the block of non-zero vectors is stored entirely in the bottom tile. Taking advantage

of the upper-triangular structure of the top tile means that a significant speed up can be acheived compared to a naiive implementation.

Applications: `DLARFB` and `DSSRFB` utilise very similar methods as the application steps of the factorisations. The product is formed by computing each column x_j of the resulting matrix $\hat{A}_{ij}$ by applying vectors forming an implicit Q^T to a column of the identity e_j in a manner similar to the application in `DGEQT2`.

The double application operation, `DSSRFB` eliminates the need to read the vectors from the diagonal tile, thus eliminates a dependency, by using the fact that the top portion of the householder vectors is an implied identity block. The following product in Eq. (11) is formed by forming each column of the result seperately:

$$\begin{pmatrix} \hat{A}_{kj} \\ \hat{A}_{ij} \end{pmatrix} = Q^T \begin{pmatrix} A_{kj} \\ A_{ij} \end{pmatrix} \tag{11}$$

C *Scheduling Design*

The tiled algorithm proceeds as a number of steps, $k = 1 \rightarrow q$, where q is the number of columns of tiles. The tasks, and the dependencies between them can be represented in a Directed Acyclic Graph, a DAG, with a node for each task and an arc where one task reads data written by another.

For tasks of the following types at step k, the dependencies are as follows:

- `DGEQT2`(A_{kk}) depends only on the diagonal tile A_{kk} being ready. If $k = 1$, there are no dependencies for this task, and as such it is the first to be executed. Otherwise, this depends on the `DSSRFB` task in A_{kk} from previous k having completed.
- `DLARFB`(V_{kk}, A_{kj}) depends on the `DGEQT2`(A_{kk}) task having completed and, if $k > 1$, again the `DSSRFB` in A_{kj} from the previous k having completed.
- `DTSQT2`(R_{kk}, A_{ik}) depends on both the `DGEQT2` of the A_{kk} portion and the `DSSRFB` of tile A_{ik} from the previous k having completed.
- `DSSRFB`(V_{ik}, A_{kj}, A_{ij}) depends on both `DLARFB` on the A_{kj} block and `DTSQT2`(R_{kk}, A_{ik}) on the A_{ik} block having completed. Also depends on the `DSSRFB` on block A_{ij} from the previous k if $k > 1$ having completed.

A crucial thing to note about these dependencies is that, despite Algorithm 2, if the tasks are executed only such their dependencies are not violated, it will result in a correct result. Also of interest is that the factorisation tasks unlock entire rows of applications, making their execution higher priority to the parallel operation of the algorithm.

The scheduling system will store the tasks in a $p \times q$ grid, because at any given time, each tile has a maximum of one task operating on it that is ready to be executed, so there is a one-to-one correspondence between each cell in the task grid and each tile. When a task is requested, the scheduler selects the highest priority task from amongst the tasks which are ready, and returns this task.

When a task is finished executing, the task's details are sent to the scheduler and any new tasks the completed tasks generates, respecting dependencies are inserted into the grid in the appropriate places. Checking what tasks are generated is calculated based on where the task is in the grid:

- If it is on the diagonal it is a DGEQT2, and generates DLARFBs along its row and the first DTSQT2.
- If it is in the column directly below the diagonal, it is a DTSQT2 and generates its row of DSSRFB and the subsequent DTSQT2
- If it is in the remaining submatrix, is is a DSSRFB and it generates the next step's tasks according to its position relative to the diagonal in this submatrix.

All of these generations, apart from the very first DGEQT2 task are subject to dependencies. If a task cannot generate another because a dependency would be violated, the task is not generated but will be generated when the task's other dependants are complete, therefore every task will be generated eventually.

To demonstrate this scheduling strategy, I will go through a whole run of the algorithm on a matrix with $p = q = 2$, making the grid of tiles 2×2.

Initially we have a DGEQT2 task in the $(1, 1)$ tile:

DGEQT2	-
-	-

Then the DGEQT2 task is completed and its details returned to the scheduler. The successor tasks DLARFB and DTSQT2 are generated:

<DONE>	DLARFB
DTSQT2	-

Now the scheduler has a choice between the two ready tasks. If we say for example that DTSQT2 is chosen and executed. Upon completion, the $(2, 2)$ DSSRFB is not inserted into the task grid because it depends on DLARFB completing, which has not happened yet. The task grid now looks like this:

<DONE>	DLARFB
<DONE>	-

DLARFB Is chosen and executed as it is the only possibility. Upon completion of the task, the $(2, 2)$ DSSRFB now has no incomplete tasks it depends on so is spawned, and the task grid takes the following form:

<DONE>	<DONE>
<DONE>	DSSRFB

The single DSSRFB is picked and executed and spawns the final task DGEQT2, yielding the final task grid:

<DONE>	<DONE>
<DONE>	DGEQT2

This final task is completed and the task grid only contains tasks which are complete, creating a finished state.

D GPU CUDA Design

The implementation of FR3 and FR4 is discussed here. A number of languages, chiefly FORTRAN, C and C++ would be suitable for implementing this project; all have CUDA support and are widely used by the scientific computing community, but I feel my familiarity with C, and its purely imperative paradigm are most suitable for this project. These implementations do focus on performance, and the extra functionality available in the C++ STL would be useful however would likely be too much of an overhead in the CUDA environment, and CUDA does not natively support class structures either, so much of the advantage in picking C++ over C is lost here.

A brief overview of the CUDA architecture: the GPU consists of multiple Streaming Microprocessors, each capable of running a small number of blocks of threads in parallel. Kernels are the function calls which make up a CUDA program, and when they are called, they are executed on a number of blocks each containing a number of threads. Each block has access to a shared memory, and threads within a block can explicitly set synchronisation points. A stream can be seen as a series of kernel invocations which execute serially. Kernel invocations on different streams can execute concurrently. A more detailed discussion of the CUDA architecture can be found in (Sanders & Kandrot 2010).

The two overarching processes the GPU has to execute are to schedule tasks and to execute tasks. These will be accomplished using different kernels operating on different streams, meaning they will execute in parallel. Since memory accesses and execution are done in units of 32 threads, warps, I will keep the number of threads equal to some multiple of 32. The matrix for computation, A, will reside in global memory on the device. The idea is to have a scheduling kernel managing a $p \times q$ matrix of tasks on the GPU.

Initially there will be one task in the task matrix, the $(0, 0)$ `DGEQT2`. This will be read, marked as executing, its data will be extracted and written to a waiting queue for an execution block. Once the block has finished executing the task, it will write the finished task to a global finished-tasks queue, where it will be picked up by the scheduler block and processed.

The scheduler will calculate the updates, if any, which are required in the task matrix and will make any required changes, distributing any newly ready tasks amongst execution blocks, where the process continues until no more tasks could be generated. When the scheduler reaches this point, it has completed and writes a null task to each execution block and both the kernels return. The resulting matrix will then be copied back to the host. A high level description of this process is summarised in Algorithms 3 & 4 for the scheduling and execution kernels respectively.

The issues surrounding this process and their solutions are as follows:

1. The number of blocks and threads in the scheduler kernel call: There will only be one block of 32 threads. There is not much scope for parallelism here, so a single warp will accomplish everything we want.

2. The number of blocks and threads in the execution kernel call: The kernel invocation for the execution kernel will be for a number of blocks equal to the number of stream microprocessors on available on the device, and the number of threads will be 32. The choice for the number of blocks is so that we can be sure that the competition for resources for each block will be minimised, and the choice for the number of threads is because our tile size will be set to 32. With 32 threads per block and 32 elements in a column of a tile, there is a one-to-one correspondence between the data and the threads.

3. Writing to a queue in a manner that is both thread-safe and non-blocking: A mechanism to achieve this with a FIFO queue on a shared memory multiprocessor system, of which a GPU is a type, has been outlined in (Tsigas & Zhang 2001). We will use a similar method to manage the task queues.

4. Equal distribution of tasks amongst blocks currently with less work than others: Deciding to which block the task should be allocated is a difficult problem, and the only way of doing it exactly is to query each execution block for its current load and give the task to the one with the least This will not be efficient due to the need to do the comparison with all blocks before deciding on one. A simple method of solving this is a round-robin approach, where each task is allocated to each block in turn: if there are 8 tasks and 4 execution blocks, each block will get 2 tasks. Unfortunately not all of the tasks here take the same time to execute so not all tasks represent equal time-slices. Double operations `DTSQT2` and `DSSRFB` each take at most twice as long as `DGEQT2` and `DLARFB`. The solution then, is a round-robin allocation, counting the "double" tasks as some amount > 1 as much as the single tasks, which will be derived empirically.

As 32 threads are available to each task, operations requiring a sum using all threads, such as the calculation of norms, can be calculated with a reduction to increase performance. Further performance gains can be had in the update of a matrix; the update of each column can be done in parallel.

Algorithm 4 Scheduling Kernel

```
read DGEQT2 (0, 0) from task matrix
write DGEQT2 (0, 0) to block 0 ready queue
while not complete do
    read finished queue until task available
    calculate changes to task matrix based on finished task
    decide whether complete or not
    apply changes to task matrix
    distribute ready tasks amongst blocks
end while
write null tasks to all blocks
```

Algorithm 5 Execution Kernel

```
read ready queue for this block until task becomes available
while first element of ready queue not null-task do
    calculate type of task and appropriate offsets into matrix A
    execute task on A depending on task type
    write task to finished queue
    read ready queue for this block until task becomes available
end while
```

References

Blumofe, R. D., Joerg, C. F., Kuszmaul, B. C., Leiserson, C. E., Randall, K. H. & Zhou, Y. (1995), Cilk: An efficient multithreaded runtime system, *in* 'Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)', Santa Barbara, California, pp. 207–216.

Buttari, A., Dongarra, J., Kurzak, J., Langou, J. & Tomov, S. (2006), The impact of multicore on math software, *in* 'In PARA 2006, UmeËŽa Sweden'.

Buttari, A., Langou, J., Kurzak, J. & Dongarra, J. (2008), 'Parallel tiled qr factorization for multicore architectures', *Concurrency and Computation: Practice and Experience* **20**(13), 1573–1590.

Buttari, A., Langou, J., Kurzak, J. & Dongarra, J. (2009), 'A class of parallel tiled linear algebra algorithms for multicore architectures', *Parallel Computing* **35**(1), 38 – 53.

Golub, G. H. & van Loan, C. F. (1989), *Matrix Computations*, 2nd edn, The Johns Hopkins University Press.

Sanders, J. & Kandrot, E. (2010), *CUDA by Example: An Introduction to General-Purpose GPU Programming*, 1 edn, Addison-Wesley Professional.

Trefethen, L. N. & Bau, D. (1997), *Numerical Linear Algebra*, Society for Industrial and Applied Mathematics.

Tsigas, P. & Zhang, Y. (2001), A simple, fast and scalable non-blocking concurrent fifo queue for shared memory multiprocessor systems, *in* 'in Proceedings of the 13th ACM Symposium on Parallel Algorithms and Architectures', ACM, pp. 134–143.