A Memory-Efficient Algorithm for Large-Scale Symmetric Tridiagonal Eigenvalue Problem on Multi-GPU Systems

Hyunsu Cho and Peter A. Yoon
Department of Computer Science, Trinity College, Hartford, CT, USA

Abstract— *Divide-and-conquer algorithm is a numerically stable and efficient algorithm that computes the eigenvalues and eigenvectors of a symmetric tridiagonal matrix. We often face the situation where the input matrix fits into the main memory but not into the on-chip memory of a GPU device. We present an out-of-core implementation where only part of the input matrix is resident in GPU memory at any point in time. It works independently of the physical size of GPU memory, handling any size of input as long as it fits into the main memory. Work is dynamically allocated to multiple GPUs and CPU cores, taking account of available workspaces and progress of the algorithm. In addition, it delivers a performance comparable to that of conventional multi-GPU implementations for cases where workspaces fit into the GPU memory.*

Keywords: Symmetric eigenvalue problem, parallel computation, general-purpose GPU computing, CUDA

1. Introduction

Divide-and-conquer algorithm is a widely used algorithm that computes the eigenvalues and eigenvectors of a symmetric tridiagonal matrix. The algorithm is known to be numerically stable and efficient when computing the full spectrum of eigenvalues [1]. Furthermore, any general symmetric eigenvalue problem can be reduced to tridiagonal form via a series of orthogonal similarity transformations. When combined with a deflation step, the algorithm delivers a good overall performance: it takes about $O(n^{2.3})$ flops to compute all the eigenvalues and eigenvectors of an $n \times n$ matrix [2].

The idea of using multiple GPUs to handle large matrices is not new. In particular, MAGMA library [3], [4] features a hybrid implementation of divide-and-conquer that uses both multiple GPUs and multicore CPUs. It off-loads the most costly portion of the algorithm, matrix multiplication, to the GPUs. Each GPU memory stores a part of the workspace, which is periodically synchronized with its counterpart in the main memory. This approach works well most of the time on multi-GPU systems, as intermediate workspaces do not grow beyond the total memory of all the GPU devices installed. Unfortunately, for very large input matrices, intermediate matrices may fit into the main memory but still exceed the total size of GPU memory. This situation may arise because GPU memory is limited in size compared to main memory. For instance, one NVIDIA® Tesla® K20c supports only about 5 GB of memory. Intermediate workspaces still have to be loaded to GPU memory, so that GPU cores can make high-bandwidth accesses.

We overcome this limitation by fixing the size of GPU workspaces to be less than available GPU memory. Depending on the size of GPU memory and that of the main memory, we dynamically compute the partition for block matrix multiplication. With fixed GPU workspaces, we are free to deal with any large input matrices, as long as the input matrix fits into the main memory. We confirmed that our implementation could handle input size as large as 50,000×50,000.

The overhead required by dynamic partition can be problematic for small subproblems. A general criterion is whether a subproblem fits entirely into a single GPU's memory. For small problems, it is better to avoid block matrix multiplication entirely. Instead, we let GPU devices solve multiple subproblems in parallel. This has an additional benefit of hiding latency in memory transfer, which is relatively costly compared to the small computational work involved.

This paper is organized as follows: Section 2 presents a brief overview of divide-and-conquer algorithm for symmetric tridiagional eigenvalue problem.

Section 3 discusses how tasks should be organized in modules. Section 4 discusses important details regarding our out-of-core implementation on multi-GPU systems. Finally, Section 5 presents performance results and analysis.

2. Divide-and-conquer algorithm

Let A be an $n \times n$ symmetric tridiagonal matrix where the diagonal and subdiagonal entries are given by a_i's and b_i's respectively. The idea is to transform A into a sum of two smaller tridiagonal systems:

$$A = \left[\begin{array}{c|c} \tilde{A}_1 & \\ \hline & \tilde{A}_2 \end{array}\right] + H = \tilde{A} + H \tag{1}$$

where

$$\tilde{A}_1 = \begin{bmatrix} a_1 & b_1 & & \\ b_1 & \ddots & \ddots & \\ & \ddots & a_{m-1} & b_{m-1} \\ & & b_{m-1} & a_m - b_m \end{bmatrix}$$

$$\tilde{A}_2 = \begin{bmatrix} a_{m+1} - b_m & b_{m+1} & & \\ b_{m+1} & \ddots & \ddots & \\ & \ddots & a_{n-1} & b_{n-1} \\ & & b_{n-1} & a_n \end{bmatrix}$$

and

$$H = \left[\begin{array}{cc|cc} & & b_m & b_m \\ & & & \\ \hline b_m & & & \\ b_m & & & \end{array}\right].$$

Now that we managed to divide the given eigenvalue problem into two problems of smaller size, we can merge the eigendecompositions of $\tilde{A}_1$ and $\tilde{A}_2$ to get the eigendecomposition of $\tilde{A}$.

Suppose we have obtained the eigendecomposition of $\tilde{A}_1$ and $\tilde{A}_2$, that is, we compute orthogonal matrices $\tilde{Q}_1, \tilde{Q}_2$ and diagonal matrices $\tilde{D}_1, \tilde{D}_2$ such that

$$\tilde{A}_1 = \tilde{Q}_1 \tilde{D}_1 \tilde{Q}_1^T \text{ and } \tilde{A}_2 = \tilde{Q}_2 \tilde{D}_2 \tilde{Q}_2^T.$$

Then the eigendecomposition of $\tilde{A}$ is given by

$$\tilde{A} = \left[\begin{array}{c|c} \tilde{A}_1 & \\ \hline & \tilde{A}_2 \end{array}\right] = \tilde{Q}\tilde{D}\tilde{Q}^T$$

where

$$\tilde{Q} = \left[\begin{array}{c|c} \tilde{Q}_1 & \\ \hline & \tilde{Q}_2 \end{array}\right] \quad \text{and} \quad \tilde{D} = \left[\begin{array}{c|c} \tilde{D}_1 & \\ \hline & \tilde{D}_2 \end{array}\right].$$

The remainder of the algorithm involves transforming the eigenvalues and eigenvectors to take account of the matrix H being added on the right-hand side. To compute the eigendecomposition of A from that of $\tilde{A}$, we perform a process known as *rank-one update* [1].

The matrix H, also known as the *rank-one modifier*, is a product of form

$$H = \rho \mathbf{w}\mathbf{w}^T$$

where

$$\rho = b_m \text{ and } \mathbf{w} = \left[\begin{array}{c} \mathbf{e}_m \\ \hline \mathbf{e}_1 \end{array}\right].$$

Here, $\mathbf{e}_i$ is the i^{th} elementary unit vector. It follows that

$$\begin{aligned} A &= \tilde{Q}\tilde{D}\tilde{Q}^T + \rho \mathbf{w}\mathbf{w}^T \\ &= \tilde{Q}(\tilde{D} + \tilde{Q}^T \rho \mathbf{w}\mathbf{w}^T \tilde{Q})\tilde{Q}^T \\ &= \tilde{Q}(\tilde{D} + \rho \mathbf{z}\mathbf{z}^T)\tilde{Q}^T \end{aligned} \tag{2}$$

where

$$\mathbf{z} = \tilde{Q}^T \mathbf{w} = \begin{bmatrix} \text{last column of } \tilde{Q}_1^T \\ \text{first column of } \tilde{Q}_2^T \end{bmatrix}.$$

Thus, it suffices to compute the eigendecomposition of the matrix $\tilde{D} + \rho \mathbf{z}\mathbf{z}^T$. If $\tilde{D} + \rho \mathbf{z}\mathbf{z}^T = \hat{Q} D \hat{Q}^T$, then the eigendecomposition of A is given by

$$\begin{aligned} A &= \tilde{Q}(\tilde{D} + \rho \mathbf{z}\mathbf{z}^T)\tilde{Q}^T = \tilde{Q}\hat{Q} D \hat{Q}^T \tilde{Q}^T \\ &= Q D Q^T \end{aligned} \tag{3}$$

where $Q = \tilde{Q}\hat{Q}$.

It only remains to find the eigenvalues and eigenvectors of $\tilde{D} + \rho \mathbf{z}\mathbf{z}^T$. This task consists of three distinct subtasks:

2.1 Perform deflations

Let d_i's be the entries of the diagonal matrix D and the z_i's be the entries of the vector $\mathbf{z}$:

$$\tilde{D} = \operatorname{diag}(d_1, d_2, \cdots, d_n), \quad \mathbf{z} = [z_1 \quad z_2 \quad \cdots \quad z_n]^T.$$

It turns out that, whenever $d_i = d_{i+1}$ or $z_i = 0$ for some i, we get an eigenvalue for free: d_i itself is an eigenvalue of $\tilde{D} + \rho \mathbf{z}\mathbf{z}^T$. Furthermore, the corresponding eigenvector is either $\mathbf{e}_i$ (if $z_i = 0$) or some rotation of it (if $d_i = d_{i+1}$). This phenomenon is called *deflation*. In practice, deflations occur frequently, when $|d_i - d_{i+1}|$ or $|z_i|$ is small enough.

The major saving occurs in the matrix multiplication step in (3): we can leave out the i-th eigenvalue and eigenvector from the computation of $\hat{Q}$ [2]. Instead, we infer their values directly from $\tilde{Q}$, which is already available. Hence, we skip the corresponding rows and columns when we compute $Q = \tilde{Q}\hat{Q}$. In this way, matrix multiplication in (3) can be accelerated so that the whole algorithm costs only $O(n^{2.3})$ in time instead of $O(n^3)$.

Let $T + \rho \mathbf{u}\mathbf{u}^T$ be the submatrix that is the result of deflating the matrix $\tilde{D} + \rho \mathbf{z}\mathbf{z}^T$:

$$T = \operatorname{diag}(\delta_1, \delta_2, \cdots, \delta_k) \text{ and } \mathbf{u} = [\zeta_1 \quad \zeta_2 \quad \cdots \quad \zeta_k]^T$$

where $\delta_1 < \delta_2 < \cdots < \delta_k$ and $\zeta_i \neq 0$ for all i.

2.2 Computing the eigenvalues via the secular equation

Let λ be an eigenvalue of $T + \rho \mathbf{u}\mathbf{u}^T$ with an associated eigenvector $\mathbf{q}$. Then by definition,

$$(T + \rho \mathbf{u}\mathbf{u}^T)\mathbf{q} = \lambda \mathbf{q}, \tag{4}$$

so that

$$T\mathbf{q} + \rho(\mathbf{u}^T \mathbf{q})\mathbf{u} = \lambda \mathbf{q} \tag{5}$$

It turns out that $\mathbf{u}^T \mathbf{q} \neq 0$; otherwise, $\lambda = \delta_i$ for some i and $\zeta_i = 0$, which contradicts the conditions of $T + \rho \mathbf{u}\mathbf{u}^T$. Similarly, we conclude that $\lambda \neq \delta_i$ for all $i = 1, \cdots, k$.

Since $\lambda \neq \delta_i$ for all i, the diagonal matrix $T - \lambda I$ has no zero entry and its inverse is well defined. With some algebra, it is possible to show that (5) is equivalent to

$$1 + \rho \mathbf{u}^T (T - \lambda I)^{-1} \mathbf{u} = 0 \tag{6}$$

This equation is equivalent to a rational equation known as *the secular equation*:

$$1 + \rho \sum_{i=1}^{n} \frac{\zeta_i^2}{\delta_i - \lambda} = 0 \tag{7}$$

The k solutions of the secular equation give the eigenvalues of $T + \rho\mathbf{u}\mathbf{u}^T$. The equation can be solved by a variant of the Newton-Raphson method in which approximating lines are replaced with approximating rational asymptotes. Li [5] lays out the full details of a secular equation solver and offer solutions to common issues in numerical stability of the algorithm.

2.3 Computing the eigenvectors

Once we obtain the eigenvalues λ_i of $T + \rho\mathbf{u}\mathbf{u}^T$, we compute the corresponding eigenvectors $\mathbf{q}_i$. In theory, $(T - \lambda_i I)^{-1}\mathbf{u}$ gives an eigenvector of λ_i:

$$\begin{aligned}&(T + \rho\mathbf{u}\mathbf{u}^T)[(T - \lambda I)^{-1}\mathbf{u}]\\ &= ((T - \lambda I) + \lambda I + \rho\mathbf{u}\mathbf{u}^T)[(T - \lambda I)^{-1}\mathbf{u}]\\ &= \lambda[(T - \lambda I)^{-1}\mathbf{u}] \end{aligned} \tag{8}$$

Unfortunately, two computed eigenvectors are not numerically orthogonal whenever their associated eigenvalues are close to each other. Gu and Eisenstat [6] proposed a more stable way to compute numerically orthogonal eigenvectors of $T+\rho\mathbf{u}\mathbf{u}^T$. In a nutshell, their approach amounts to solving an *inverse eigenvalue problem*: Let $\lambda_1, \lambda_2, \cdots, \lambda_k$ be the roots of the secular equation (7). Let $\hat{\mathbf{u}}$ be the vector whose k entries are given by

$$\hat{u}_i = \sqrt{\frac{\prod_{j=1}^{k}(\lambda_j - \delta_i)}{\rho \prod_{\substack{j=1\\ j\neq i}}^{k}(\delta_j - \delta_i)}} \tag{9}$$

Also, let

$$\Lambda = \operatorname{diag}(\lambda_1, \lambda_2, \cdots, \lambda_k).$$

Then the matrix $\Lambda + \hat{\mathbf{u}}\hat{\mathbf{u}}^T$ has $\lambda_1, \lambda_2, \cdots, \lambda_k$ as its eigenvalues. Furthermore, $(\Lambda - \lambda_i I)^{-1}\hat{\mathbf{u}}$ gives a numerically stable eigenvector of each eigenvalue λ_i.

3. Task organization

Using LAPACK routine `dstedc` as a guide [7], we organize divide-and-conquer algorithm in the following modules:

- `dlaed0`: Split the given problem into 128×128 subproblems, performing appropriate rank-one cuts. Then compute the eigendecomposition of each 128×128 subproblem by calling the QR routine `dsteqr`. Finally, let `dlaed1` merge the eigendecompositions of adjacent submatrices until we have the eigendecomposition of the original matrix.
- `dlaed1`: Coordinate subtasks necessary to merge the eigendecompositions of two adjacent submatrices. Specifically,
  - Produce $\tilde{D} + \rho\mathbf{z}\mathbf{z}^T$ from A via (2).
  - Call `dlaed2` to carry out Subtask 2.1
  - Call `dlaed3` to carry out Subtasks 2.2 and 2.3.
  - *Back-transform* the eigenvector collection $\hat{Q}$ of $\tilde{D} + \rho\mathbf{z}\mathbf{z}^T$ by multiplying with $\tilde{Q}$, as described in (3).
- `dlaed2`: Perform deflation as given by Subtask 2.1. To differentiate between deflated eigenvalues and eigenvectors from non-deflated ones, we maintain an ordered list of eigenvalues [7]. Each time we deflate an eigenvalue, we remove it from the ordered list and put it at the end of the list; in other words, we permute the list. Let σ be the permutation that results from deflation.
- `dlaed3`: Compute the eigendecomposition $\tilde{D} + \rho\mathbf{z}\mathbf{z}^T = \hat{Q}D\hat{Q}^T$ by carrying out Subtasks 2.2 and 2.3. Now that all the deflated eigenvalues are at the end of the list, we can focus on the non-deflated portion, i.e. $T + \rho\mathbf{u}\mathbf{u}^T$. The $\mathbf{u}$ vector is given by $\mathbf{z}$ with σ applied. More specifically, `dlaed3` does the following steps:
  - Call `dlaed4` to compute each root λ_i of the secular equation.
  - Solve the inverse eigenvalue problem to compute numerically orthogonal eigenvectors that correspond to λ_i's.

  After `dlaed3` returns, `dlaed1` should re-merge the deflated eigenvalues back into the middle of the list.
- `dlaed4`: Compute the i-th root λ_i of the secular equation. We use an iteration scheme known as the Middle Way, where we create a series of approximating rational functions whose asymptotic poles match those of the secular equation near λ_i [5].

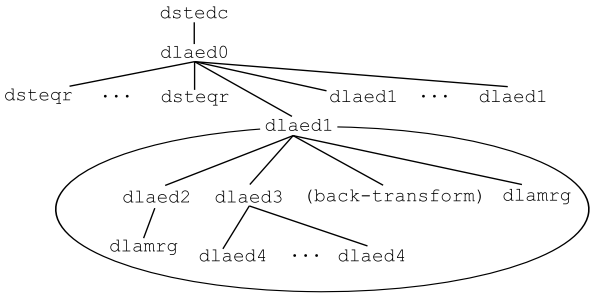

Fig. 1: A call graph of `dstedc`.

4. Parallel Implementation on Multiple GPUs

The key idea is to fix the size of GPU workspaces so that we do not run out of GPU memory regardless of the size of input matrices. Now the only limiting factor is the main memory, which in many systems is in plentiful supply. Since the input can be of any size but GPU workspaces are not, it is crucial to build a dynamic partition of tasks. More importantly, the nature of work changes as the algorithm progresses.

Like other algorithms of its kind, divide-and-conquer algorithm starts with many small base cases (cf. Fig. 2).

As small systems are merged into larger ones, there would be fewer and fewer subproblems left. In other words, the work at hand gradually becomes more coarse-grained. Thus, we need to adapt the way we allocate tasks depending on the current size of subproblems.

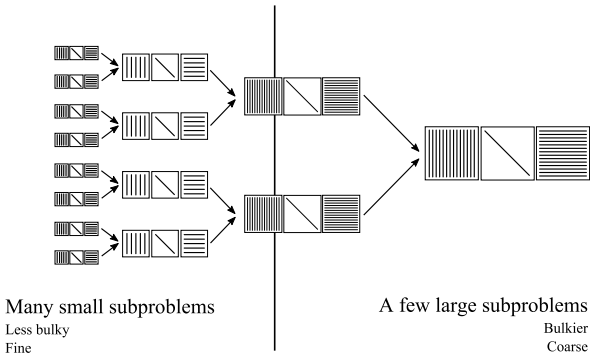

Fig. 2: A schematic of divide-and-conquer algorithm

Before we discuss dynamic allocation of tasks, let us briefly look at CUDA™, a general-purpose GPU computing platform.

4.1 CUDA programming environment

Graphical processing units (GPUs) are commodity hardware that were originally designed to accelerate graphics applications. In recent years, a number of non-graphical, computationally expensive algorithms have been implemented on GPUs [8]. In particular, NVIDIA offers a general-purpose API called CUDA™. All recent NVIDIA graphics cards support this interface.

GPUs are massively parallel processors in which many small worker threads execute in parallel. While each thread may not be as powerful as a typical CPU core, the collaboration of many threads helps achieve a high throughput. GPUs follow a data parallelism paradigm in which each worker thread executes a similar set of instructions but processes its own portion of data.

In typical circumstances, a GPU does not launch its own work. Instead, a CPU thread launches a *kernel*, or a subroutine, to be executed on a selected GPU. The CUDA runtime launches multiple instances of the kernel to be run by the GPU threads. Kernel launch parameters determine how many GPU threads are launched and how they are organized.

A defining characteristic of GPU programming is that GPUs have memory spaces separate from the main memory. GPUs cannot access the main memory directly; instead, content has to be copied from the main memory to the GPU memory first. This step is essential in supporting a large degree of parallelism, as the GPU processing cores require a dedicated memory designed for high bandwidth. The data transfer passes through the PCI Express channel, making the operation relatively costly. Furthermore, the size of GPU memory is also a limit; even high-end models carry only a few gigabytes of dedicated memory. To make matters worse, on systems with multiple graphics cards installed, the GPUs have memory spaces separate from one another. Thus, a CPU thread that copies a buffer into a GPU memory needs to designate a specific target GPU.

4.2 Dynamic block partition of back-transform

A computational bottleneck in divide-and-conquer algorithm is the back-transformation step at the end of `dlaed1`. When only a few eigenvalues deflate, its cost approaches $O(n^3)$ flops, where n is the number of eigenvalues. Fortunately, this step is a BLAS 3 operation and scales well on GPUs. It is where MAGMA makes most use of GPUs [3], and we intend to do so as well.

Given an $n \times k$ transformation matrix $\tilde{Q}$ and a $k \times k$ collection $\hat{Q}$ of eigenvectors, the transformed eigenvectors are given by the product $\tilde{Q}\hat{Q}$. Both n and k change over time, n being the size of subproblems at the current level and k being the number of non-deflated eigenvalues. Let G be the greatest integer such that three $G \times G$ matrices fit into a single GPU device's memory. Let D be the number of GPU devices installed. The idea is to pick a multiple of D that is large enough so that

$$\frac{n}{aD} \leq G.$$

Then a block matrix of dimension $n/aD \times k/aD$ will certainly fit into a single GPU device. Let A_{ij} and B_{ij} be block matrices of $\tilde{Q}$ and $\hat{Q}$, respectively, where each A_{ij} is $n/aD \times k/aD$ and each B_{ij} is $k/aD \times k/aD$. We now have a conformable partition of matrix multiplication.

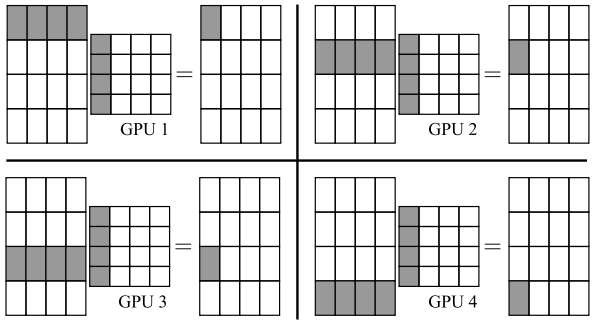

Fig. 3: Out-of-core block multiplication using 4 GPUs

Since we partitioned the matrix product in multiples of D, it is straightforward to assign block multiplications to the D GPU devices (cf. Fig. 3). Notice that no more than one A block and one B block need to be resident in each GPU device at any moment. Once each partial product $A_{ip}B_{pj}$ $(1 \leq i,\ j,\ p \leq aD)$ is computed, the corresponding block C_{ij} can be incremented by that amount. Matrix multiplication is supported by cuBLAS [9], a fast GPU implementation of BLAS interface.

4.3 Fine-grained parallelism

Out-of-core matrix multiplication is fairly inefficient when the operands are small — there is a constant overhead of moving block matrices back and forth between the GPU and CPU memories. In addition, we have to compute the matrix partition for each subproblem. If we could keep everything in one place, we would be able to eliminate all the overhead.

To hide the overhead, we let the GPUs solve multiple subproblems in parallel, each GPU solving one subproblem. The benefit of such approach is two-fold. First, we avoid performing out-of-core matrix multiplication when it is not necessary. Second, we hide the latency of data transfer by overlaying multiple subproblems on top of each other. For instance, at the moment when GPU 1 is fetching a workspace from the main memory, GPU 2 may be decomposing another matrix. Fig. 4 shows a visual representation of overlapping merge tasks. The CUDA toolkit incorporates a visual profiler capable of drawing a timeline of kernel launches [10].

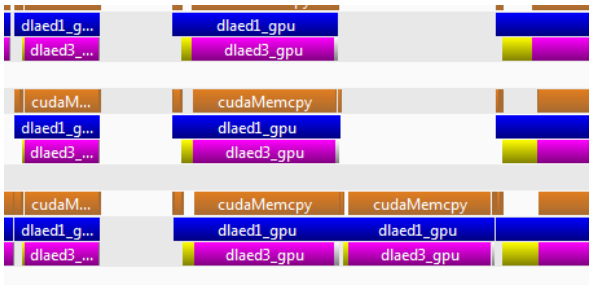

Fig. 4: Overlapping of multiple merge tasks

Unfortunately, certain parts of divide-and-conquer do not scale well on GPUs. Especially, deflation process involves construction of permutations and has to be done serially. We solve this problem by paring each GPU device with a host thread and forming a compute group. On the other hand, both secular equations and inverse eigenvalue problems can be solved efficiently in bulk parallel fashion by GPUs: each λ_i can be computed independent of other λ_j's, and similarly with the eigenvectors.

4.4 Profiling

Extending the idea of simultaneous merging, we also make use of idle CPU cores and form compute groups as well. An added difficulty is that performance scales at different rates on GPU-CPU groups and on CPU-only groups. We overcome this difficulty by constructing linear regression models of respective compute groups.

Consider GPU-CPU groups first. We define two independent variables that affect performance: let X_1 be the subproblem size and X_2 be the number of GPU-CPU groups. The dependent variable is Y, the time it takes to solve a subproblem of size X_1 using X_2 groups. We model their relationship by a power function of form

$$Y = X_1^{\alpha_1} X_2^{\alpha_2} 2^{\alpha_3}.$$

where $\alpha_1, \alpha_2, \alpha_3$ are parameters to be fitted. Similarly, let X_3 be the number of CPU-only groups and Z be the time it takes for X_3 CPU groups to solve subproblem of size X_1:

$$Z = X_1^{\beta_1} X_3^{\beta_2} 2^{\beta_3}.$$

The models reflect our intuition to some degree: for instance, if performance were to scale linearly with respect to the number of groups, Y would be proportional to X_2^{-1}, suggesting $\alpha_2 \approx -1$. In addition, the $O(n^{2.3})$ work complexity of the algorithm suggests that the scaling of Y is some multiple of that of X_1.

Each of the models is nonlinear on its own, but we can easily transform it into a linear model. Taking the logarithm of both sides gives

$$\log Y = \alpha_1 \log X_1 + \alpha_2 \log X_2 + \alpha_3$$
$$\log Z = \beta_1 \log X_1 + \beta_3 \log X_3 + \beta_3.$$

Now the parameters can be fitted using the method of least squares. Given the parameters, we estimate the ratio R between performance of GPU-CPU groups and that of CPU-only groups:

$$R = \frac{Z}{Y} = X_1^{\beta_1 - \alpha_1} X_3^{\beta_2} X_2^{-\alpha_2} 2^{\beta_3 - \alpha_3}$$

The ratio enables our implementation to balance loads by allocating the right number of subproblems to each kind of compute groups. Our code package incorporates a separate profiler that runs test matrices and computes the parameters. It saves the parameters to a configuration file so that the main subroutine could load them at startup.

5. Performance

Our machine comprises a dual 2.0 GHz Intel® Xeon® E5-2620 CPU and four NVIDIA® Tesla® K20c graphics cards. The machine was configured with 64 GB main memory and 5 GB memory for each GPU. Our experiments used double-precision floating point arithmetic. A package containing the full source code and the performance profiler is available at `https://github.com/hcho3/dstedc_mgpu`.

Prior work such as [3] show that the empirical complexity of divide-and-conquer algorithm depends on the characteristics of the input matrix. If a significant portion of the eigenvalues of a subsystem deflate out, the cost is closer to $O(n^2)$ rather than $O(n^3)$. For the purpose of this experiment, we choose a simple random sample $\lambda_1, \cdots, \lambda_n$ from the standard normal distribution and generate a symmetric tridiagonal matrix whose eigenvalues are λ_i's. For all the test matrices we generated this way, 8-12% of eigenvalues deflate.

Despite the limited amount of memory available on GPU devices, our implementation was able to handle up to 50,000×50,000 input matrices, for which outputs and workspaces combined occupied 85% of the main memory. On the GPU side, only 3.9 GB out of 5 GB was used. However,

MAGMA's implementation could not handle input matrices larger than 36,000. Fig. 5 illustrates how our implementation handles large matrices stably even in the face of limited GPU memory.

Table 1: Performance for various test matrices

		Performance (sec)		
Matrix dimension		Hybrid	CPUs only	Speedup
In-core	1024	0.98	0.34	0.35
	2048	1.80	0.93	0.52
	4096	3.84	3.83	1.00
	8192	9.28	17.43	1.88
Out-of-core	16384	26.91	99.80	3.71
	32768	103.00	681.56	6.62
	36000	117.38	867.70	7.39
	50000	239.36	2278.90	9.52

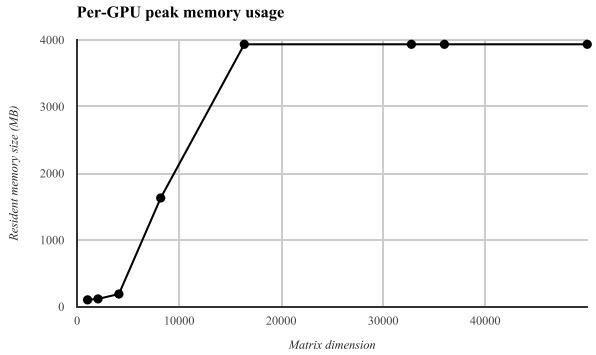

Fig. 5: Average GPU memory consumption for different input sizes

To put our implementation's performance in context, we created a version that exclusively uses CPU cores (cf. Table 1). For smallest input matrices, the CPU-only version shows better performance. One significant factor is that, unlike GPUs, CPU cores share the same memory space. So when the algorithm progresses from one level to next, it is possible to re-group the CPU cores to form fewer compute groups. Also, the overhead of setting up multiple CUDA contexts is absent.

On the other hand, the hybrid version does better for 8192×8192 input matrices and larger, as the back-transformation step takes a growing share of flops. At the same time, matrix multiplication is a bulk parallel task and scales well on GPUs. The performance profile is illuminating in that regard: the values of α_i's and β_i's were respectively

$$\alpha_1 = 0.978, \alpha_2 = -0.916, \alpha_3 = -11.884$$
$$\beta_1 = 2.401, \beta_2 = -0.529, \beta_3 = -26.788.$$

This means that each time the subproblem size was doubled, GPU-CPU groups spent only twice as much time as it had, whereas CPU-only groups had to spend 5.3 times as much. The model produced a good fit for the data points of the profile, giving R-squared coefficients of 0.984 and 0.996 for GPU-CPU groups and CPU-only groups, respectively.

6. Conclusion

In this paper, we presented a memory-efficient implementation of divide-and-conquer algorithm on multi-GPU systems. Our implementation made use of both multiple GPUs and multicore CPUs. We overcame the limitations in GPU memory by fixing GPU workspaces to a size independent of subproblem size. This approach allowed our implementation to handle input matrices as large as 50,000×50,000.

Furthermore, despite the added complexity caused by the fixed size of GPU workspaces, our implementation exhibited a significant speedup for large input matrices compared to a version that used multicore CPUs exclusively. At the same time, we allocated tasks for the fine-grained portion of the algorithm. By solving multiple subproblems simultaneously, some on GPUs and some on CPUs, our implementation solve small problems at a rate comparable to the case where only CPUs are used.

References

[1] D. S. Watkins, *Fundamentals of Matrix Computations* (New York: John Wiley and Sons Incs., 1991).
[2] J. Demmel, *Applied Numerical Linear Algebra* (Philadelphia: SIAM, 1997).
[3] C. Vomel, S. Tomov and J. Dongarra, Divide and conquer on hybrid GPU-accelerated multicore systems, *SIAM Journal on Scientific Computing*, 34(2), 2012, C70-C82
[4] MAGMA Library, version 1.4.1. URL: http://icl.cs.utk.edu/magma/
[5] R-C. Li, Solving secular equations stably and efficiently, LAPACK Working Notes, 1994.
[6] M. Gu, S.C. Eisenstat, A stable and efficient algorithm for the rank-one modification of the symmetric eigenproblem, *SIAM Journal on Matrix Analysis and Applications*, 15(4), 1994, 1266-1276
[7] J. Rutter, A serial implementation of Cuppen's divide and conquer algorithm for the symmetric eigenvalue problem, LAPACK Working Notes, 1994.
[8] D. B. Kirk and W. W. Hwu, *Programming Massively Parallel Processors: A Hands-on Approach* (Burlington, MA : Morgan Kaufmann Publishers, 2010).
[9] NVIDIA, cuBLAS Library User Guide. http://docs.nvidia.com/cuda/pdf/ CUBLAS_Library.pdf.
[10] NVIDIA, Profiler User's Guide. http://docs.nvidia.com/cuda/pdf/ CUDA_Profiler_Users_Guide.pdf
[11] J. Dongarra, T. Dong, M. Gates, A. Haidar, S. Tomov, I. Yamazaki, MAGMA: a new generation of linear algebra library for GPU and multicore architectures, *Supercomputing*, Salt Lake City, Utah, 2012.
[12] I. Yamazaki, T. Dong, R. Solca, S. Tomov, J. Dongarra and T. Schulthess, Tridiagonalization of a dense symmetric matrix on multiple GPUs and its application to symmetric eigenvalue problems, *Concurrency and Computation: Practice and Experience*, published online, 2013, DOI: 10.1002/cpe.3152
[13] NVIDIA, CUDA C Programming Guide.
[14] E. Agullo, J. Demmel, J. Dongarra, B. Hadri, J. Kurzak, J. Langou, H. Ltaief, P. Luszczek and S. Tomov, Numerical linear algebra on emerging architectures: The PLASMA and MAGMA projects, *Journal of Physics: Conference Series*, 180(1), 2009, 12-37.