Using GPUs to Accelerate the Bisection Algorithm for Finding Eigenvalues of Symmetric Tridiagonal Matrices

Vasily Volkov
James Demmel

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2007-179
http://www.eecs.berkeley.edu/Pubs/TechRpts/2007/EECS-2007-179.html

December 29, 2007

Acknowledgement

We want to thank ATI and NVIDIA for the donated GPUs, Takahiro Katagiri for providing his implementation of the Multi-section with Multiple Eigenvalues method and Professor Sara McMains for the course on general-purpose computation on the GPUs and being helpful with equipment.

Using GPUs to Accelerate the Bisection Algorithm for Finding Eigenvalues of Symmetric Tridiagonal Matrices

Vasily Volkov

Computer Science Division
University of California at Berkeley

James W. Demmel

Computer Science Division and Department of Mathematics
University of California at Berkeley

Abstract

Graphical Processing Units (GPUs) potentially promise widespread and inexpensive high performance computation. However, architectural limitations (only some operations and memory access patterns can be performed quickly, partial support for IEEE floating point arithmetic) make it necessary to change existing algorithms to attain high performance and correctness. Here we show how to make the bisection algorithm for eigenvalues of symmetric tridiagonal matrices (`sstebz` from LAPACK) run both fast and correctly on an ATI Radeon X1900 GPU. Our fastest algorithm takes up to 156× less time than Intel's Math Kernel Library version of `sstebz` running on the CPU, but does so by doing many redundant floating point operations compared to the CPU version. We use an automatic tuning procedure analogous to ATLAS or PHiPAC to decide the optimal redundancy. Correctness despite partial IEEE floating point semantics required explicitly adding 0 in the inner loop. The problems and solutions discussed here are of interest on other GPU architectures.

1 Motivation and Objectives

Modern graphics processors (GPUs) are data parallel architectures that can run general-purpose computations in single precision (so far) at high computational rates. They are capable of achieving 110 GFLOPS in matrix-matrix multiplication [Segal and Peercy 2006] and show 30-40x speedups compared to the recent Intel Xeon processors in computationally intensive applications such as Black-Scholes option pricing [McCool et al. 2006] and gas dynamics solvers [Hagen et al. 2007]. It is tempting to exploit this computational power in solving other common numerical problems.

In this work we consider an implementation of another widely used linear algebra routine — the bisection algorithm for finding the eigenvalues of symmetric tridiagonal matrices. A numerically robust, vectorized implementation of this algorithm in single precision is available in LAPACK's `sstebz` routine [Anderson et al. 1999]. Our goal is to port the vectorized segments of the code to the GPU. In order to increase the utilization of the parallel resources, we use the Multi-section with Multiple Eigenvalues method used previously by Katagiri et al. [2006].

For the purpose of this study we restrict our attention to finding all eigenvalues of the matrix. The extension to finding a subset of the eigenvalues as done in LAPACK's `sstebz` routine, is straightforward.

2 The Bisection Algorithm

2.1 Overview

A detailed description of the bisection algorithm can be found in Demmel [1997] or Parlett [1980]. A thorough analysis of its correctness in finite-precision machine arithmetic is presented in Demmel et al. [1995]. In the following we summarize the important features of the algorithm and present two novel techniques to ensure correctness in an unusual floating-point semantic.

```
    function Count(x)
        Count = 0
        d = 1
        for i = 1 to n
            d = a_i – x – b^2_{i–1}/d
(*)         if (d < 0) then Count = Count + 1
        endfor
```

Figure 1: The kernel of bisection algorithm. Function $Count(x)$ may be evaluated for an array of arguments concurrently.

Let T be an $n \times n$ symmetric tridiagonal matrix with diagonals $a_1, \ldots, a_n$ and off-diagonals $b_1, \ldots, b_{n-1}$. For the convenience of presentation, let $b_0 = b_n = 0$. Then the algorithm $Count(x)$ in Fig. 1 implements LDL^T decomposition of $T - xI$ without pivoting and counts the number of negative entries in the diagonal matrix D. According to Sylvester's inertia theorem it gives the number of eigenvalues of T that are less than the real number x.

Now, suppose we are given a set of 4-tuples (l_i, u_i, nl_i, nu_i) for $i = 1, \ldots, EL$ so that $nl_i = Count(l_i) < nu_i = Count(u_i)$ and the union of intervals $[l_i, u_i)$ contains all eigenvalues. Then $nu_i - nl_i$ gives the number of eigenvalues in the interval $[l_i, u_i)$. Computing $nm_i = Count(m_i)$ for $m_i = (l_i + u_i)/2$ equal to the midpoints produces new tuples (l_i, m_i, nl_i, nm_i) and (m_i, u_i, nm_i, nu_i) for half as wide intervals. Intervals that contain no eigenvalues ($nl_i = nu_i$) are discarded and the process is repeated until a sufficiently small enclosing interval for each eigenvalue is found. The interval to start the iterations is constructed using Gershgorin's theorem.

The total work done in $Count(x)$ to find k eigenvalues is $O(nk)$. This is usually much larger than $O(k)$ work done in the rest of the algorithm. This motivates the efforts in finding the more efficient implementation of $Count(x)$.

A trivial way to speedup $Count(x)$ given parallel resources is to evaluate values $nm_i = Count(m_i)$ for $i = 1, \ldots, EL$ concurrently. The utilization of parallel resources may be low unless EL is large enough. A well-known technique to increase the utilization and cut the running time of the bisection algorithm is to subdivide each interval $[l_i, u_i)$ with multiple points $m_{ij} = l_i + j(u_i - l_i) / (ML + 1)$ for $j = 1, \ldots, ML$ [Lo et al. 1987; Simon 1989; Katagiri et al. 2006]. In this case $Count(x)$ is evaluated at $EL \times ML$ points concurrently achieving better utilization of the parallel resources. $ML = 1$ corresponds to the bisection algorithm and $ML \geq 2$ is called multisection. ML is chosen to balance the gain from achieving higher utilization and the loss from introducing arithmetic redundancy by using multiple points ($ML = 1$ minimizes the operation count).

There are known alternative designs that were not considered in the present work. Newton's or Zeroin algorithm may improve convergence of intervals that are found to contain only one eigenvalue. A vectorizable alternative to $Count(x)$ such as considered by Lo et al. [1987] may allow achieving high utilization with lower arithmetic redundancy. Versions of $Count(x)$ using "parallel-prefix" to parallelize evaluations for a single x were analyzed by Ren [1996] and Mathias [1995] and found to be numerically unstable, and will not be further considered here.

pivmin	**if** ($\|d\| < pivmin$) **then** $d = -pivmin$ **if** ($d < 0$) **then** $Count = Count + 1$
SignBit	$Count = Count + SignBit(d)$
IEEE	(matrix is preprocessed by setting $a_i = a_i + 0$) **if** ($d < 0$) **then** $Count = Count + 1$
IEEE0	$d = d + 0$ **if** ($d < 0$) **then** $Count = Count + 1$

Figure 2: Possible modifications of *Count*(*x*) routine to handle overflow. The lines are used instead of (*) in Fig. 1. The pivmin version is used in LAPACK, SignBit version is used in ScaLAPACK [Blackford et al. 1997a], IEEE version is used in our CPU code and IEEE0 version is used in our GPU code.

2.2 Handling Divisions by Zero and Overflow

At a finite number of points, such as $x = a_1$, $n \geq 2$, the algorithm in Fig. 1 encounters a division by zero. Similarly, when run in machine arithmetic it may also encounter overflow. The designs of the function *Count*(*x*) outlined below produce a consistent result even at these points.

The pivmin version of *Count*(*x*) in Fig. 2 avoids divisions by zero and overflow by initially scaling the matrix (not shown) so that its largest entry is neither too close to the underflow or the overflow threshold (in particular, so that no b_i^2 overflows), and by moving the pivot *d* away from zero by a small threshold *pivmin*. This threshold is trivially computed for every input matrix. This algorithm produces a correct result in nearly every machine arithmetic. It is considered in more detail in [Demmel et al. 1995; Kahan 1966].

The SignBit version in Fig. 2 may yield faster code if the machine arithmetic supports IEEE exception handling rules and they do not incur high performance penalty [Demmel and Li 1994]. The *SignBit*(*d*) function returns 1 for negative values including $-\infty$ and –0; it returns 0 otherwise. Note that *SignBit*(*d*) differentiates –0 and +0, unlike the floating point comparison $d < 0$ done according to the IEEE standard [IEEE 1987, Ch. 5.7]. This variant requires that b_i^2 neither underflows nor overflows. Otherwise, a NaN may be produced that makes the result incorrect (tiny b_i may be set to zero, splitting the matrix into independent subproblems).

If the machine arithmetic does not provide an inexpensive way to compute *SignBit*(*d*), the function can be replaced by comparison with zero ($d < 0$) if every possibility for $d = -0$ is eliminated [Demmel et al. 1995, Ch. 8]. If machine arithmetic never produces –0 in addition, it is sufficient to preprocess all $a_i = -0$ into 0 (IEEE version in Fig. 2). If this is not the case, e.g. if underflow in addition may result in –0, we may instead similarly process the pivot *d* at every iteration as shown in the IEEE0 variant in Fig. 2. Both IEEE and IEEE0 variants require that $(-0) + 0 = 0$ holds.

IEEE0 version also produces correct result when zero's sign is occasionally lost in computation.

2.3 Monotonicity of *Count*(*x*)

In exact arithmetic, *Count*(*x*) grows monotonically with *x*. This is not necessarily true in finite precision arithmetic. It can be proven that each of the overflow-safe versions of *Count*(*x*) in Fig. 2 is monotonic if and only if the floating point arithmetic is monotonic [Kahan 1966; Demmel et al. 1995], i.e. the result of operations $x + y$, $x - y$ and x / y monotonically depends on the arguments. We note that the proof extends to the case where x / y is implemented by $x * (1 / y)$, and multiplication and reciprocation are both monotonic. IEEE floating point semantics would guarantee monotonicity.

Non-monotonic *Count*(*x*) makes the obvious implementation of the algorithm incorrect. For example, some of the intervals may be found to contain negative numbers of eigenvalues. It was shown by Demmel et al. [1995] that the bisection algorithm can be made correct in the absence of monotonicity by careful adjustment of the return values of *Count*(*x*), so that at any point in the algorithm the set of nonempty tuples (l_i, u_i, nl_i, nu_i) maintained by the algorithm satisfies 2 properties: (1) the intervals $[l_i,u_i)$ are pairwise disjoint, and (2) for each interval $Count(l_i) \leq nl_i < nu_i \leq Count(u_i)$, i.e. the values of nl_i and nu_i may be "adjusted" from their nominal values of $Count(l_i)$ and $Count(u_i)$. It can be adapted for the multisection algorithm as follows.

Let $m_1 \leq m_2 \leq \ldots \leq m_{ML}$ be the multisection points subdividing $[l, u)$. For convenience, let $m_0 = l$, $m_{ML+1} = u$ and $nm_0 = nl$, $nm_{ML+1} = nu$. Then the adjustment proceeds from $k = 1$ up to *ML* by setting

$$nm_k = \max(nm_{k-1}, \min(Count(m_k), nu)).$$

Now, 4-tuples (m_i, m_{i+1}, nm_i, nm_{i+1}) for $i = 0, \ldots, ML$ represent the finer intervals. It is easy to see, that either $nm_i = nm_{i+1}$ and the interval is discarded, or $Count(m_i) \leq nm_i < nm_{i+1} \leq Count(m_{i+1})$. This condition is sufficient for the algorithm to be correct, see Chapter 7 in Demmel et al. [1995].

2.4 Heterogeneous Computing

Heterogeneous computing environments with different rounding properties in different functional units present a hazard for correct parallel execution of bisection [Blackford et al. 1997b; Demmel et al. 1995]. However, if only function *Count*(*x*) is computed using the heterogeneous parallel resources, this threat is easily countered.

Indeed, interleaving the values of *Count*(*x*) that are evaluated on processors that use different floating point rounding rules, such as the CPU and the GPU, may result in non-monotonic behavior, even if values on each processor are monotonic. In that case the adjustment procedure described in the Section 2.3 may be used to enforce monotonicity. This resolves the problem completely.

3 The GPU Architecture

The target platform for our GPU implementation is the Radeon X1900 XT that was released by ATI in January 2006. In this section we briefly review its architectural features that are important for understanding the GPU performance. Further details can be found in vendor's technical publications [ATI 2005a; ATI 2005b; ATI 2005c; Riguer 2006; Segal and Peercy 2006; AMD 2006].

3.1 Processing Units

The GPU has an array of processors that are operated in SIMD lockstep mode. Multithreading is used to hide memory access latency. The basic data format is a four-component vector of 32-bit IEEE floating point numbers. There are three types of units that are operating in parallel at 625 MHz clock rate — ALUs, memory fetch units and flow control units.

There are 48 ALUs. Each ALU consists of a vector and a scalar unit that are programmed separately. The vector unit executes simple instructions on the first three components of the registers. These instructions include multiply and add (MAD), dot products, minimum or maximum of two values, taking fractional parts and conditional assignments. The scalar unit may execute the same instructions except the dot products on the fourth component of the registers. In addition it can compute exponent, logarithm, sine, cosine, $1 / x$ (RCP) and $1 / \sqrt{x}$ operations. Each arithmetic instruction may have an input and an output modifier. The input modifier can be $-x$, $|x|$ and $-|x|$. In addition, one of the inputs can be a result of a simple operation over two other inputs (called "presubtract"). This operation can be $x + y$, $x - y$, $1 - x$ and $1 - 2x$. The four components of the input registers can be arbitrarily shuffled and swizzled. The output modifier allows multiplying the result of

Operation Type	How Implemented	Theoretical Peak Rate
1/A	RCP	30 Gflop/s
A/B	RCP in the scalar pipe, MAD in the vector pipe	30 Gflop/s
A*B	MAD	120 Gflop/s
A+B	MAD and presubtract	240 Gflop/s
A*B+C	MAD	240 Gflop/s
(A+B)*C+C	MAD and presubtract	360 Gflop/s
(1–2*A)*B+C	MAD and presubtract	480 Gflop/s
((1–2*A)*B+C)/8	MAD, presubtract and output modifier	600 Gflop/s

Table 1: Theoretical peaks for various arithmetic operations on the Radeon X1900 XT (single precision only).

the operation by factors of 2, 4, 8, 1/2, 1/4 and 1/8. Also, the output value may be clamped to the range [0, 1]. Each unit can execute one instruction every clock cycle. The estimates of the theoretical peak performance based on this data are given in Table 1.

There are 16 memory fetch units, i.e. one per three ALUs. GPUBench [Buck et al. 2004], which is a popular benchmarking utility, shows that the cost of fetching a four-component vector is 12 ALU cycles if the data is in the cache, i.e. it takes one cycle for a unit to fetch one 32-bit floating point number. It accounts to 40 GB/s. Fetch-4 extension allows quadrupling this rate to 160 GB/s. Memory fetches that miss the cache are more expensive.

Flow control units allow implementing if-else statements by executing both branches and masking off the non-participating processors. Also they support loops that are repeated a constant number of times. The constant is limited to the range [0, 255] and can be changed only when a program is not running. A larger number of iterations may be achieved using nested loops.

3.2 Floating Point Arithmetic

The ATI CTM guide [AMD 2006] gives the most detailed specification of the GPU floating point arithmetic that we know. Still, it is not exhaustive and even disagrees with our tests done in DirectX 9.0.

Most arithmetic operations are "accurate to within one bit on each input; transcendental functions have larger tolerances", but rounding rules are not specified. Hence, it is unclear if the arithmetic is monotonic or not. We were able to determine that RCP operation is monotonic by executing it for every possible input. This is possible since it takes only one 32-bit argument, i.e. there are fewer than 2^{32} different inputs. Addition and multiplication are likely to be monotonic as their most straightforward implementations are. In our tests we observed rounding towards zero in MAD operation, which is monotonic.

The special values, such as ±∞, ±0 and NaN are supported. Though the ATI CTM guide claims that they are treated according to the IEEE 754 standard in many operations including MAD and RCP, we found that on the GPU $0*\infty = 0$ and $0*\text{NaN} = 0$, which deviates from the standard. According to our experience handling of the special IEEE values does not incur a performance penalty. Denormalized numbers are accepted but are always flushed to zero with the sign preserved. That means that underflow in addition may produce –0 that was verified by our tests. This behavior also disagrees with the IEEE standard [IEEE 1987, Ch. 6.3].

The convention $(-0) + 0 = +0$ held in our tests.

It is unclear if the adder in the presubtract units implements the same or different floating point semantics. It cannot be programmed explicitly in the high-level languages such as DirectX's HLSL.

The sign of zero may be lost when copying data between registers. The reason is the lack of a separate MOV operation, so that another arithmetic operation must be used instead. If the assignment is done using MAD, i.e. as $y = 0 * 0 + x$ or $y = 1 * x + 0$, the sign of zero is lost in the addition. A safer solution is $y = \text{MAX}(x, x)$. However the choice might not be under control when programming in a high-level language.

To summarize, underflow in addition may result in –0 on this particular GPU so that IEEE version of *Count*(*x*) may give incorrect values in corner cases. IEEE0 and pivmin versions are expected to be correct. Arithmetic is likely to be monotonic, but we cannot say it is for sure.

4 Implementation

Our implementation is done in C++ following the FORTRAN codes of the `sstebz` routine in LAPACK version 3.1 that is available in the public domain. It was compiled using the Intel C++ compiler version 9.1 with all optimizations for speed turned on except the floating point arithmetic options. In order to ensure sufficient IEEE compliance we used `/fp:source` and `/fp:except-` compiler options.

The function *Count*(*x*) is implemented both on the CPU and the GPU. In each iteration of the algorithm one of them is chosen using a performance model as elaborated in Section 4.6. The return values of *Count*(*x*) are adjusted to handle non-monotonicity following Section 2.3. There are two sources of non-monotonicity: potentially non-monotonic GPU arithmetic and mixing the results of *Count*(*x*) computed on the CPU and the GPU.

4.1 The GPU Program

The GPU is programmed using DirectX 9.0. Three different versions of *Count*(*x*) are programmed in HLSL as shown in Fig. 3. The HLSL program solves for four different values of *x* at once to fit better to the GPU architecture.

The negative pivots in the GPU program are counted using floating point numbers. This limits the counter and hence the dimension of the input matrix to $2^{24} \approx 16{,}000{,}000$. It is a sufficiently large number — it may take a few weeks to solve a problem of such a size on a Pentium 4 processor.

The loop is naturally unrolled four times as the matrix entries are fetched in four-component vectors. Unrolled four times more it yields a perceivable speedup. In order to handle arbitrary matrix dimensions, the matrix size is rounded up to the nearest multiple of four introducing dummy entries at the tail. They are set to $+\infty$ for the diagonal and to 0 for the off-diagonal entries. That could produce an error if run in IEEE-compliant arithmetic as it gives $b_i^2/d = 0/0 = \text{NaN}$ for $d = 0$, but is correct on this GPU since it implements $0*\infty = 0$. The dummy entries produce $d = +\infty$ so that *Count* is not incremented. An analogous technique would work on an IEEE-compliant platform.

The matrix is laid out into two 2D arrays ("textures") and is transferred to the GPU memory once per `sstebz` call. The iteration through the entries is implemented in two nested loops, each iterating through a horizontal or a vertical dimension of the arrays. Fetch-4 extension is not enabled for these matrix structures, since the program is computation-bound anyway (see Section 4.3).

4.2 Transferring Data and Running the Job

As the input and output data for each parallel run of *Count*(*x*) is produced and processed in the main memory, we need to transfer it to the GPU memory and back for each call to the routine.

```
     a = tex2D( matrix_a, pos );
     bb = tex2D( matrix_bb, pos );
     pos.x += increment;
(1)  d = a.R – x – bb.R / d;
(4)  Count += (d < 0) ? 1 : 0;
     d = a.G – x – bb.G / d;
     Count += (d < 0) ? 1 : 0;
     d = a.B – x – bb.B / d;
     Count += (d < 0) ? 1 : 0;
     d = a.A – x – bb.A / d;
     Count += (d < 0) ? 1 : 0;
```

(a) IEEE version in HLSL language.

```
(2) d = (|d| < pivmin) ? –pivmin : d;
(3) d = d + 0;
```

(b) Extra lines used in other versions. Line (2) is used in the pivmin version, line (3) — in IEEE0 version.

```
(1) RCP r0.R, r0.R
(1) RCP r0.G, r0.G
(1) RCP r0.B, r0.B
(1) RCP r0.A, r0.A
(1) ADD r1, r3.R, -r2
(1) MAD r0, r4.R, -r0, r1
(2) ADD r1, r0_abs, -c0
(2) CMP r0, r1, r0, -c0
(3) ADD r0, r0, c1.R
(4) CMP r1, r0, c1.R, c1.G
(4) ADD r5, r5, r1
```

```
r0: d
r1: temp
r2: x
r3: a
r4: bb
r5: Count

c0: pivmin
c1: (0,1,0,0)
```

(c) A simple mapping to PS 3.0 language.

Figure 3: *Count*(*x*) expressed in HLSL and PS 3.0. The looping and initialization logic is not shown. Variables `d` and `x` are four component vectors, each component (`R`, `G`, `B` and `A`) stays for a different instance of *Count*(*x*). Numbers to the left show how HLSL and PS 3.0 codes are related to each other.

DirectX 9.0 has limited functionality for transferring data from the GPU to the main memory. Arrays that accept the output of the programs run on the GPUs ("render targets") must be transferred entirely even if a small portion of them is needed. To avoid a performance penalty due to these extra bandwidth requirements, we create a number of render targets and choose the smallest that fits the data. The width of every render target is 64 for the reasons explained later. Heights are from 1 to 512, which allows handling as large problems as n = 131,072. The spacing between heights varies from 1 for small heights to 64 for larger for economical use of the GPU memory. The GPU program is then executed for the first H lines of the selected render target (by drawing an appropriate triangle; all possible triangles are set in the GPU memory during the initialization stage), i.e. for $64\times4\times H$ instances of *Count*(*x*). H is chosen to be the smallest that satisfies $64\times4\times H \geq EL\times ML$.

For storing an array of x (shifts) on the GPU we use a single texture also of width 64, created with D3DUSAGE_DYNAMIC flag. This flag allows updating the texture partially when transferring data to the GPU memory.

4.3 Performance of *Count*(*x*) on the GPU

Fig. 3(c) shows a simple PS 3.0 code similar to that produced by the HLSL compiler. This code approximates the machine code that is ultimately produced but not available when programming in DirectX. Note, that the two if-statements in the pivmin version in Fig. 2 do not require branches in the PS 3.0 code. Table 2

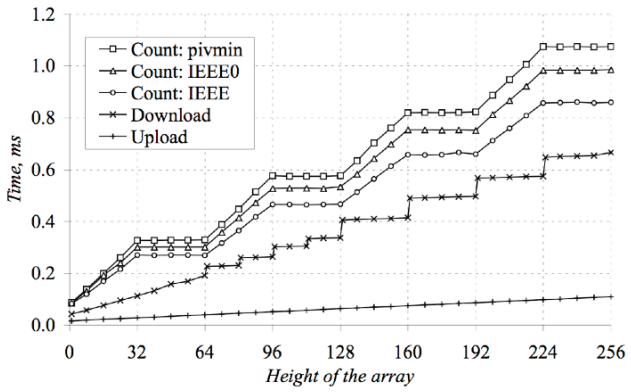

Figure 4: Running times of different GPU stages and versions of *Count*(*x*) for n = 192. Upload is the transfer to the GPU memory, and download is other way around. For height H, $256\times H$ instances of *Count*(*x*) are run.

shows the theoretical estimates of the performance using the architectural details presented in Section 3. We assume that the most of the memory fetches hit cache and are completely overlapped with computation, hence free. The number of floating point operations per iteration is 3 in each of the versions of *Count*(*x*).

The observed rates are shown in the same Table. They match the predicted rates within 10%. The code achieved up to 82% of the theoretical peak of the input bandwidth of 40 GB/s. Also, the codes performed at nearly the same rates when memory fetches have been removed from the inner loop and substituted with register assignments that preserve the data dependence pattern. This supports our assumption that the memory fetches do not incur extra latency.

Version	Clocks	Theory		Measured	
		Gflop/s	GB/s	Gflop/s	GB/s
IEEE	8	45	30	49	33
IEEE0	9	40	27	43	29
pivmin	10	36	24	38	25

Table 2: Predicted and observed computational rates and bandwidths for different GPU versions of *Count*(*x*).

Fig. 4 shows the running times of the three versions of the *Count*(*x*) and data transfers. The DirectX Query mechanism was used to wait until the GPU completes execution. The IEEE0 and pivmin versions are correspondingly 15% and 30% slower than the IEEE version that is similar to the results reported by Demmel and Li [1994]. Upload time is negligible compared to the download time (the amount of data transferred is nearly the same). At n = 192 pictured in Figure the total time spent in the data transfer is about the same as the time spent in computation. Computation time grows linearly with n but the transfer time does not change. So, at large n the time spent in transfer is not significant.

Note the stairs of period 64 that show that computational rate is higher when H is a multiple of 64. The same applies to the width of the rendered rectangle that motivated us choosing width 64 for the render targets. For example, computing 32×32, 64×32, 32×64 and 64×64 blocks of pixels take the same time. Also, the slope in the graph for H = 1,…,32 is twice as large as for H = 64*k, k=1,2,…. This could mean that only half of the processors are utilized when $H \leq 32$ and only a quarter are used when both the height and the width are less or equal to 32. A possible model for this behavior could be execution of threads in 64x64 tiles, each tile split into four quadrants that are assigned to different processors. If the quadrant is empty, the processors it is assigned to are

```
for( i = 0; i < size; i++ )
{
   d[i] = 1.0;
   Count[i] = 0.0;
}
for( j = 0; j < n; j++)
   for( i = 0; i < size; i++ )
   {
      d[i] = a[j] – x[i] – bb[j] / d[i];
      Count[i] += d[i] < 0.0 ? 1.0 : 0.0;
   }
```

Figure 5: The vectorized CPU version of the *Count*(*x*) routine that executes *size* instances of *Count*(*x*).

idle. Similar behavior was observed by Bolz et al. [2003] on NVIDIA's hardware.

In our experience a similar program run in OpenGL has demonstrated similar stairs with a period of 32. It might lead to the conclusion, that this parameter (the "stair" width) is adjustable, though the handle is not available to the programmer.

4.4 CPU Implementation of *Count*(*x*)

Similarly as it is done in LAPACK, we implemented both vectorized and non-vectorized versions of *Count*(*x*) on CPU. Both are the IEEE version, that is the fastest and correct as the CPU arithmetic is IEEE compliant (when using `/fp:source` compiler option). As in the GPU codes, floating point numbers are used to count the negative pivots. Fig. 5 shows the vectorized version of the routine. According to the compiler messages, every line in the loop body is compiled into SIMD instructions. The non-vectorized version has inverse loop order (the compiler vectorizes only inner loops). The non-vectorized version is run when $size = EL*ML < 8$.

The running time of this CPU version is labeled "naïve" in Fig. 6. As one may see, its runtime does not increase monotonically, which means that increasing *size* by computing *Count*(*x*) at a few extra points may decrease the runtime! Thus, if $size \geq 3 \pmod 8$ we add dummy points to increase *size* to the nearest multiple of 8. The new runtime is shown in the same Figure labelled "dummies". It is up to 3.2x faster according to the graph.

4.5 Testing Correctness of *Count*(*x*)

We constructed a 4×4 tridiagonal matrix, with eigenvalues sufficiently distant from zero, that yields a negative denormalized pivot in *Count*(*x*) at $x = 0$. If denormals are flushed to zero preserving sign, it produces $d = -0$. We checked if this produces a correct result in different implementations of *Count*(*x*). Note, that as all eigenvalues are far from zero, roundoff error may not influence the value of *Count*(0), as the algorithm is backward stable and the symmetric eigenvalue problem is well-conditioned.

Among the three GPU algorithms only the IEEE version has failed, as was expected in Section 3.2. Both vectorized and non-vectorized variants of CPU *Count*(*x*) produced correct results. On other hand, the results of CPU *Count*(*x*) were incorrect when the code was compiled using `/fp:fast` option, that allows achieving higher computational rates by waving strict IEEE 754 compliance.

As only the IEEE0 and pivmin versions of the GPU *Count*(*x*) are proven to be correct on this GPU and the IEEE0 version is clearly faster than the alternative, only the IEEE0 routine was used in GPU computations in the rest of the paper. IEEE and pivmin versions may still be considered when computing on other GPU models with different floating point conventions.

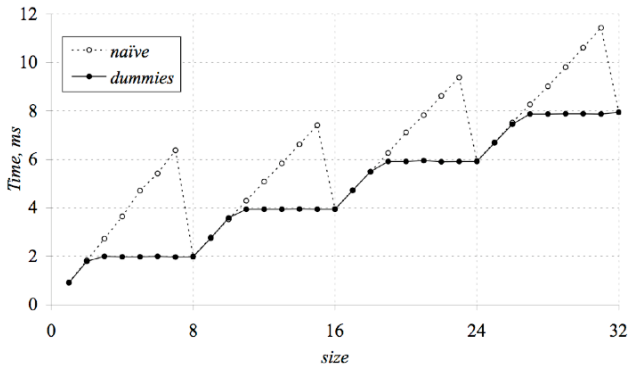

Figure 6: Running time of the CPU kernel for $n = 65536$.

4.6 Tuning

There are two choices to be made in every iteration of the bisection algorithm — what version of *Count*(*x*) to use (the GPU or the CPU one) and how large *ML* should be. The choice that results in shortest running times of the bisection algorithm should be preferred. We consider all possible choices and choose the most efficient in terms of the following definition:

$$efficiency = \frac{\log(ML+1)}{Time},$$

where *Time* is the running time of one iteration under the choice of *ML* and version of *Count*(*x*). For example, splitting each interval into 2 parts ($ML = 1$) in time T has the same efficiency as splitting it into 8 parts ($ML = 7$) in time $3T$. Finer subdivision (higher *ML*) done in the same time, and faster computation at same *ML* are considered more efficient. All decisions are made offline and tabulated for use at runtime by the CPU.

The value of *Time* is estimated using the results of a thorough benchmarking. The nonlinear stair-like behaviour of the GPU code is captured in a table with an entry for each H. A linear dependence on n is assumed ($time = latency + n * bandwidth$). Time spent outside of the *Count*(*x*) is estimated as $\alpha + \beta*EL + \gamma*ML + \delta*EL*ML$. The coefficients are fit using weighted linear least squares to minimize the relative error; the weights are set equal to the measured time. The runtimes of the CPU version of *Count*(*x*) are fit similarly, taking into account the zigzag pattern and introduction of the dummy entries.

5 Results

All results cited in this Section (and others) were obtained with 2.8 GHz Pentium 4 520 (Prescott) and ATI Radeon X1900 XT. The following implementations were used in the tests:

- CPU-alone: multisection running *Count*(*x*) on the CPU only;
- GPU-alone: multisection running *Count*(*x*) on the GPU only;
- CPU-GPU: multisection running *Count*(*x*) both on the GPU and CPU. This is our fastest code;
- CLAPACK: bisection routine `sstebz` in CLAPACK 3.0;
- MKL BZ: bisection routine `sstebz` in Intel MKL 9.0;
- MKL RF: `ssterf` routine in Intel MKL 9.0 that is QR algorithm optimized for finding all eigenvalues only;
- MKL GR: `sstegr` routine in Intel MKL 9.0 that uses dqds algorithm to find the eigenvalues only.

In routines that require specifying the absolute tolerance the value 2**sfmin* was used, that is the finest acceptable for `sstebz`. *sfmin* is the smallest value such that 1/*sfmin* does not overflow.

The following matrices were used in tests ($i = 1 \ldots n$, $\varepsilon = 2^{-23}$ is the machine epsilon):

- uniform: $a_i = 1 + (i-1)/n$, $b_i = 2/n$;

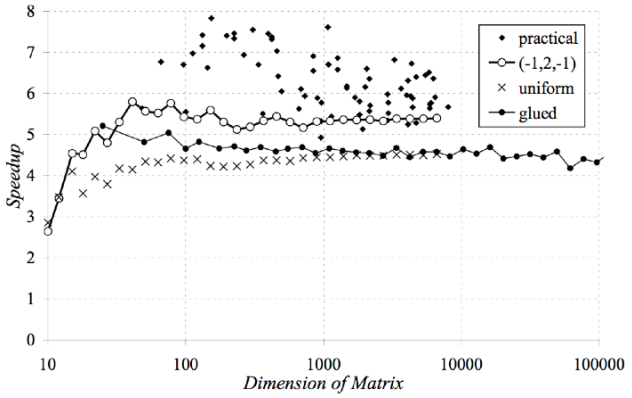

Figure 7: Speedup of our CPU version of `sstebz` relative to the version in Intel MKL 9.0.

- geometric: $a_i = (3\varepsilon)^{(i-1)/(n-1)}$, $b_i = a_{i+1}/3$;
- (–1,2,–1): $a_i = 2$, $b_i = -1$;
- glued: (–1,2,–1) matrix with $b_k = 3\varepsilon$ when $k = 0 \pmod{25}$, n is a multiple of 25;
- practical: a subset of matrices from Harwell-Boeing, University of Florida and George Fann collections reduced to tridiagonal form[1].

Uniform and geometric matrices approximate uniform and geometric distribution of eigenvalues respectively. The glued matrix has eigenvalues strongly clustered around the eigenvalues of the (–1,2,–1) matrix with $n = 25$.

Off-diagonals of the test matrices were always large enough that the LAPACK `sstebz` routine does not break the problem into smaller ones, which is not currently implemented in our version. We also ensured that other algorithms used (such as dqds) do not exhibit unusually fast convergence that happens when the matrix is very close to diagonal. This explains the choice of entries such as 3ε above.

First we analyze the behavior of the CPU codes alone. Fig. 7 shows the speedup achieved in our CPU code relative to the MKL version. It ranges from 4.2 to 7.8 for $n > 50$. The CLAPACK version was from 5% faster to 25% slower than MKL version.

Fig. 8 shows the computational rates in CLAPACK, CPU-alone and CPU-GPU versions. Only the floating point operations in *Count*(x) algorithm were taken into account in this data. CLAPACK performs at 76–182 Mflop/s and our CPU-alone version is at 270–800 Mflop/s. CPU-GPU version shows up to 40 Gflop/s, which is up to 50 — 220 times higher than the peak rates in the CPU-alone — CLAPACK versions respectively. However, the CPU-GPU version did up to 7.6× more flops than the bisection algorithm in CLAPACK and *ML* used was up to 85. For comparison, the largest optimal *ML* used in the CPU-only and GPU-only versions was 8 and 1024 respectively.

To understand the importance of using multisection vs. bisection, we performed runs forcing *ML* = 1. Fig. 9 shows that GPU-alone version is sped up by the factors of 5 to 6.6 for $n < 100$ by using multisection. Speedups at large n are substantial only if the eigenvalues are clustered — the speedup was about 4.2× for the glued matrices. Speedup should also be substantial when finding only a small subset of all eigenvalues, which is currently not implemented. Speedup is less noticeable in the GPU-CPU version as it runs on the CPU whenever the GPU multisection is too slow — the speedup was only up to 2.0×.

Fig. 10 compares the runtimes of CPU-only, GPU-only and CPU-GPU versions for the (–1,2,–1) matrix. The runtime of the

[1] available at http://crd.lbl.gov/~osni/Codes/stetester/

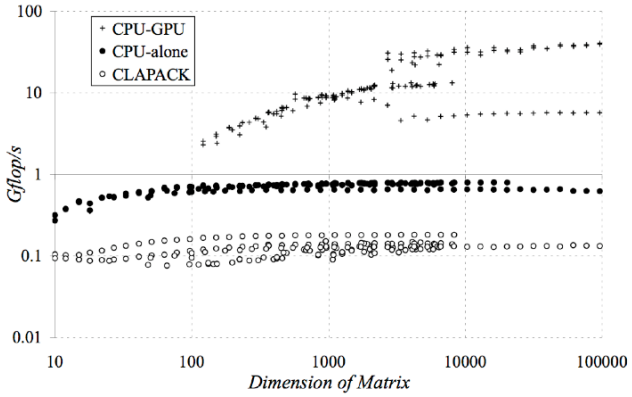

Figure 8: Computational rates (Gflops/s) achieved by different versions of `sstebz` and a mix of matrices.

Figure 9: Speedup gained in the GPU-alone version by using multisection vs. bisection.

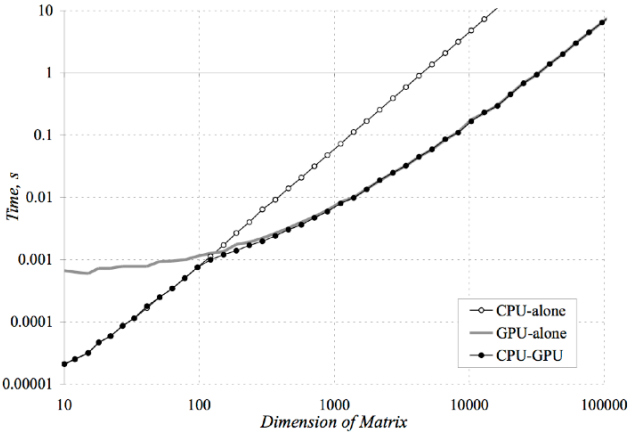

Figure 10: The comparison of the runtimes for the case of (–1,2,–1) matrix.

CPU-GPU version is nearly the minimum of the runtimes of the other two versions. The crossover between CPU-alone and GPU-alone versions is at $n \approx 130$. For $n = 1000$ the GPU-alone version is ≈ 9.0× faster than the CPU-alone version.

Fig. 11 shows the percentage of time spent in *Count*(x) on the CPU and on the GPU in the CPU-GPU version for (–1,2,–1) matrix. At $n = 121$ the time spent in the GPU code jumps from 0% to 67% and time spent in the CPU code falls from 90% to 21%. 90% and 99% of the time spent in the GPU codes are reached at $n \approx$ 1400 and $n \approx 62000$ respectively.

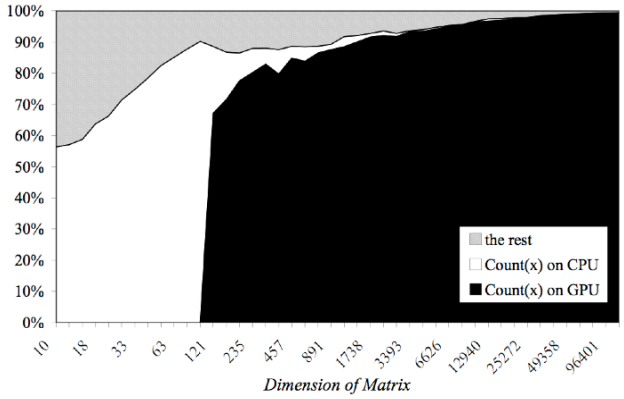

Figure 11: The breakdown of the runtime of CPU-GPU version for (–1,2, –1) matrix. The time cited for *Count*(*x*) on the GPU includes transferring data between the GPU and the main memories.

The runtimes and accuracy achieved in tests with different matrices and algorithms are shown in Fig. 13–15 and Tables 3–4. The CPU-GPU version was up to 156× faster than MKL BZ version and up to 45× faster than CPU-only version. Alternative eigensolvers were also substantially outperformed: for practical matrices the CPU-GPU version was up to 68× faster than MKL RF and up to 137× faster than MKL GR.

λ_i^{true} in Table 4 was found using LAPACK `dstebz` routine in the Intel MKL that is the double precision implementation of the bisection algorithm. According to the table, the MKL RF and MKL GR solvers showed substantially lower accuracy than the implementations of the bisection algorithm. All implementations of the bisection algorithms have shown similar absolute accuracy. The relative error defined as $\max_i (|\lambda_i^{computed} - \lambda_i^{true}| / |\varepsilon\ \lambda_i^{true}|)$ for geometric matrices was also small — 1.49, 1.97 and 1.33 for CPU-only, CPU-GPU and MKL BZ implementations correspondingly. This shows that the GPU-based solver has high relative accuracy that is expected in some special cases, see [Barlow and Demmel 1990].

We found that *Count*(*x*) in CPU-alone and GPU-alone versions was always monotonic in the tests, but CPU-GPU version did produce non-monotonic values, requiring the correction discussed in Sections 2.3 and 2.4.

We also tried running *Count*(*x*) concurrently on the GPU and the CPU but found that this does not yield substantial benefits as the GPU usually outperforms the CPU by at least an order of magnitude.

6 Comparison with Previous Work

NVIDIA CUDA 1.1 SDK contains another implementation of bisection algorithm that is optimized for NVIDIA's GPUs [Lessig 2007]. This implementation suffers from many overflow problems in data structures that leads to failures or crashed when $512 \le n \le 1024$, and failures with geometric matrices with $n > 1024$ that yield severely imbalanced interval trees. Also, it has a non-practical stopping criterion and is correct only if GPU arithmetic is monotonic, that is not clear given the detailed vendor's programming guide [NVIDIA 2007].

However, we managed to successfully run this code with uniform matrices and stopping criterion aligned with that in LAPACK as done in our implementation. Running it on GeForce 8800 GTX, which is a newer and faster GPU than was used in our work, it performed up to more than 2 times slower than our code run on Radeon X1900 XT and Pentium 4, see Fig. 12. For example, our code runs in 1.0s for n=40000 vs. 2.5s for the NVIDIA's code.

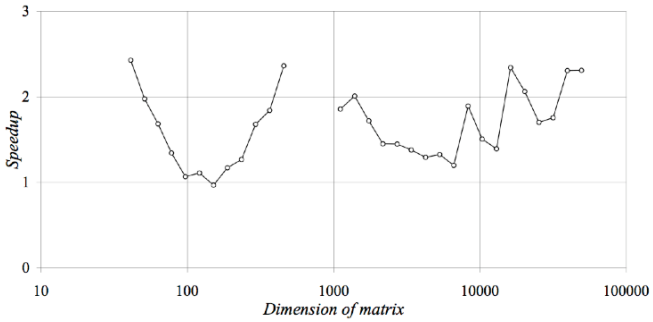

Figure 12: Speedup of our code run on Radeon X1900 XT vs. the code in CUDA SDK 1.1 run on GeForce 8800 GTX.

Matrix	CPU-only	MKL BZ	MKL RF	MKL GR
practical	22	125	68	137
(–1,2–1)	38	130	38	89
uniform	45	156	37	89
geometric	31	107	21	52
glued	2.9	12.8	780	670

Table 3: Maximum slowdowns of different implementations relative to the CPU-GPU version.

Matrix	CPU-only	CPU-GPU	MKL BZ	MKL RF	MKL GR
practical	1.31	1.28	1.31	38	148
(–1,2,–1)	1.01	1.01	1.00	24	28
uniform	1.00	1.00	1.00	630	260
geometric	1.23	1.25	1.24	112	79
glued	1.00	1.00	1.00	57	109

Table 4: Worst absolute errors observed in tests. Absolute error is defined as $(\max_i |\lambda_i^{computed} - \lambda_i^{true}|) / (\varepsilon \max_i |\lambda_i^{true}|)$.

NVIDIA's implementation runs entirely on the GPU. This required a significant programming effort that is described in [Lessig 2007], as the rest of the algorithm beyond *Count*(*x*) is not embarrassingly parallel. In our opinion, there is little motivation for this complicated and error-prone design, since *Count*(*x*) dominates the cost for sufficiently large problems, say, takes 90% of time for $n > 100$ as in Fig. 11. On other hand, if the problem is small, it is faster to solve it entirely on the CPU, see Fig. 10. Another argument for putting the entire algorithm on the GPU is to avoid the communication overhead at each call to *Count*(*x*). But as it was shown in Section 4.3 this overhead is not substantial when run on Radeon X1900 for sufficiently large matrices. It may even be less substantial with newer GPUs as they have an order or magnitude higher bandwidths in the CPU-to-GPU transfers.

As the GPU usually comes with a CPU (and in the future may come on the same die, as is discussed today by both Intel and AMD), we advocate departing from the trend of moving entire algorithms to the GPU to considering instead the CPU-GPU tandem as the target platform. Many existing parallel algorithms spend small fraction of the work in codes that do not expose substantial parallelism. Offloading this work to the GPU may be both painful and unprofitable.

7 Conclusion

We have produced a numerically correct implementation of the bisection algorithm for the GPU that substantially outperforms the bisection and other algorithms run on the CPU. Automatic tuning was one of the key components of our high performance design. We took advantage of the partial compliance of the GPU arithmetic with the IEEE 754 standard to reduce the runtime about 15%.

Also, we showed that a higher degree of IEEE 754 compliance could win an additional 15% assuming no performance penalty for greater compliance. Trivial improvement would raise the functionality of our implementation to the full functionality of LAPACK's `sstebz`, such as finding only a subset of eigenvalues and splitting the matrix into blocks for better performance when off-diagonal elements are small. Future work includes porting a tridiagonal eigenvector solver, such as the MRRR algorithm [Dhillon and Parlett 2003] or the inverse iteration algorithm. Using the GPU in the reduction to tridiagonal form promises speedup in the dense symmetric eigenproblems — these algorithms are rich in BLAS2 and BLAS3 operations such as matrix multiply, which is known to run faster on the GPU.

Acknowledgements

We want to thank ATI and NVIDIA for the donated GPUs, Takahiro Katagiri for providing his implementation of the Multi-section with Multiple Eigenvalues method and Professor Sara McMains for the course on general-purpose computation on the GPUs and being helpful with equipment.

References

AMD 2006. *ATI CTM Guide, version 1.01*.

ATI 2005a. *Radeon X1000 Family Technology Overview*, ATI Technology White Paper.

ATI 2005b. *Radeon X1800 Memory Controller*, ATI Technology White Paper.

ATI 2005c. *Radeon X1800 Shader Architecture*, ATI Technology White Paper.

ANDERSON, E., BAI, Z., BISCHOF, C., DEMMEL, J., DONGARRA, J., DU CROZ, J., GREENBAUM, A., HAMMARLING, S., MCKENNEY, A., OSTROUCHOV, S., AND SORENSEN, D. 1999. *LAPACK Users' Guide*, SIAM.

BARLOW, J., AND DEMMEL, J. 1990. Computing accurate eigensystems of scaled diagonally dominant matrices, *SIAM Journal on Numerical Analysis 27*, 3, 762–791. (Also LAPACK Working Note #7).

BLACKFORD, L. S., CHOI, J., CLEARY, A, D'AZEVADO, E., DEMMEL, J., DHILLON, I., DONGARRA, J., HAMMARLING, S., HENRY, G., PETITET, A., STANLEY, K., WALKER, D., AND WHALEY, R. 1997a. *ScaLAPACK Users' Guide*, SIAM.

BLACKFORD, L. S., CLEARY, A, DEMMEL, J., DHILLON, I., DONGARRA, J., HAMMARLING, S., PETITET, A., REN, H., STANLEY, K., AND WHALEY, R. 1997b. Practical experience in the numerical dangers of heterogeneous computing, *ACM Transactions on Mathematical Software 23*, 2, 133–147. (See also LAPACK Working Note #112)

BOLZ, J., FARMER, I., GRINSPUN, E., AND SCHRÖDER, P. 2003. Sparse matrix solvers on the GPU: conjugate gradients and multigrid, *ACM Transactions on Graphics 22*, 3, 917–924.

BUCK, I., FATAHALIAN, K., AND HANRAHAN, P. 2004. GPUBench: evaluating GPU performance for numerical and scientific applications, *ACM Workshop on General Purpose Computing on Graphics Processors (GP^2)*.

DEMMEL, J. W. 1997. *Applied Numerical Linear Algebra,* SIAM.

DEMMEL, J. W., DHILLON, I., AND REN, H. 1995. On the correctness of some bisection-like parallel algorithms in floating point arithmetic, *Electronic Transactions on Numerical Analysis 3*, 116–149. (Also LAPACK Working Note #70).

DEMMEL, J. W., AND LI, X. 1994. Faster numerical algorithms via exception handling, *IEEE Transactions on Computers 43*, 8, 983–992.

DHILLON, I., AND PARLETT, B. N. 2003. Orthogonal eigenvectors and relative gaps, *SIAM Journal on Matrix Analysis and Applications 25*, 3, 858–899.

HAGEN, T. R., HENRIKSEN, M. O., HJELMERVIK, J. M., AND LIE, K.-A. 2007. How to solve systems of conservation laws numerically using the graphics processor as a high-performance computational engine, In *Geometrical Modeling, Numerical Simulation and Optimization: Industrial Mathematics at SINTEF*, Eds., Hasle, G., Lie, K.-A., and Quak, E., Springer Verlag, 211–264.

IEEE 1987. IEEE standard 754-1985 for binary floating-point arithmetic, *SIGPLAN 22*, 2, 9–25.

KAHAN, W. 1966. *Accurate Eigenvalues of a Symmetric Tri-Diagonal Matrix*, Technical Report CS41, Computer Science Department, Stanford University, July 22, 1966 (with revisions to June 1968).

KATAGIRI, T., VÖMEL, C., AND DEMMEL, J. W. 2006. Automatic performance tuning for the multi-section with multiple eigenvalues method for the symmetric eigenproblem, In *PARA'06*, Umea, Sweden, June, 2006.

LESSIG, C. 2007. *Eigenvalue Computation with CUDA*, NVIDIA CUDA SDK 1.1.

LO, S.-S., PHILIPPE, B., AND SAMEH, A. 1987. A multiprocessor algorithm for the symmetric tridiagonal eigenvalue problem, *SIAM Journal on Scientific and Statistical Computing 8*, 2, 155–165.

MATHIAS, R. 1995. The instability of parallel prefix matrix multiplication, *SIAM Journal of Scientific Computing 16*, 4, 956–973.

MCCOOL, M., WADLEIGH, K., HENDERSON, B., AND LIN, H.-Y. 2006. *Performance Evaluation of GPUs Using the RapidMind Development Platform*, October 26, 2006.

NVIDIA 2007. *NVIDIA CUDA Compute Unified Device Architecture Programming Guide, version 1.1*.

PARLETT, B. N. 1980. *The Symmetric Eigenvalue Problem*, Prentice-Hall.

REN, H. 1996. *On the Error Analysis and Implementation of Some Eigenvalue Decomposition and Singular Value Decomposition Algorithms*, PhD Thesis in Applied Mathematics, University of California at Berkeley (see also LAPACK Working Note #115).

RIGUER, G. 2006. *The Radeon X1000 Series Programming Guide*, Radeon SDK, March 2006.

SEGAL, M., AND PEERCY, M. 2006. A performance-oriented data parallel virtual machine for GPUs, *ACM SIGGRAPH 2006 Sketches*.

SIMON, H. D. 1989. Bisection is not optimal on vector processors, *SIAM Journal on Scientific and Statistical Computing 10*, 1, 205–209.

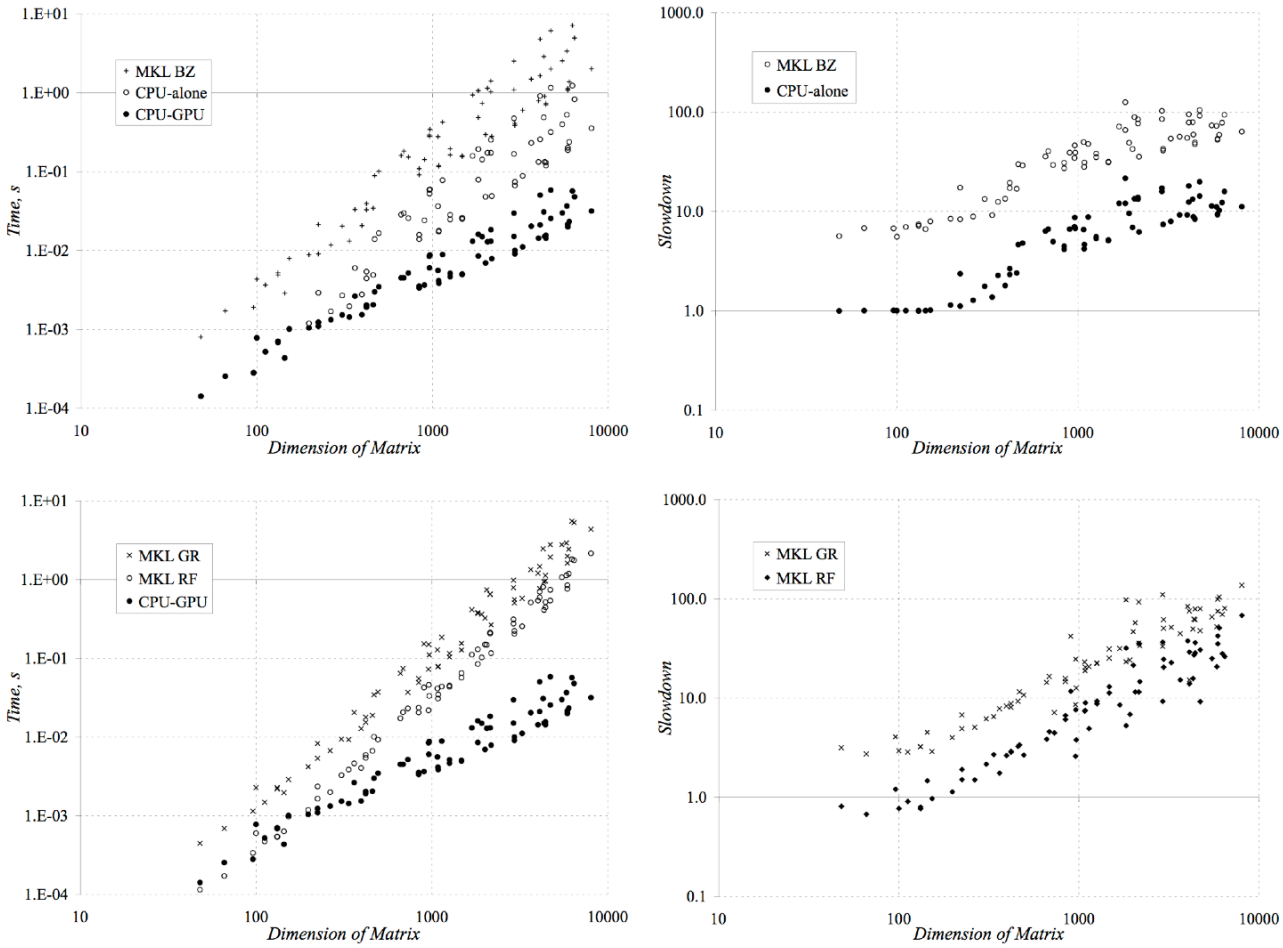

Figure 13: Practical matrices: runtimes of different implementations and slowdowns relative to the CPU-GPU version.

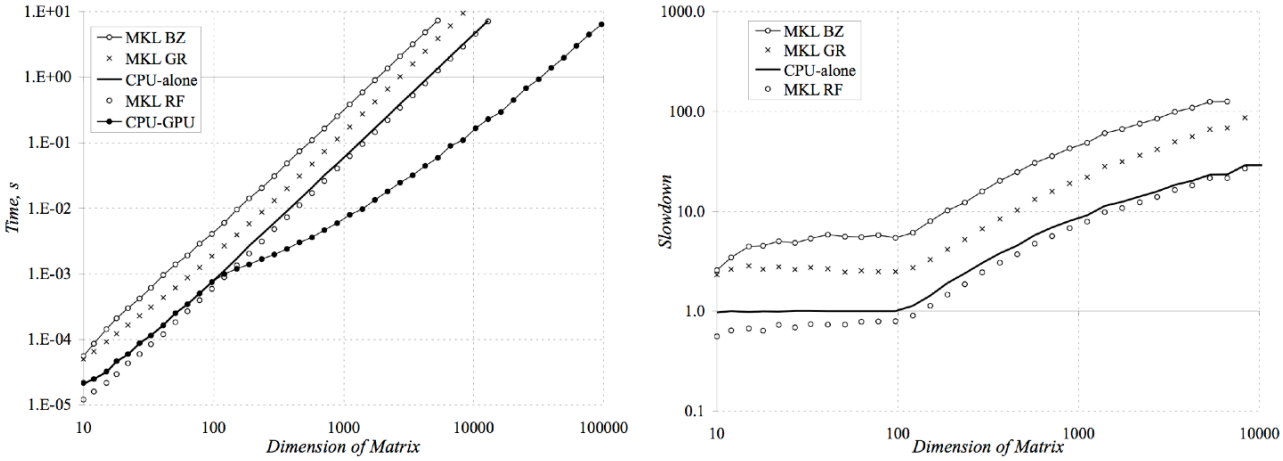

Figure 14: (–1,2,–1) matrices: runtimes of different implementations and their slowdowns relative to the CPU-GPU version. Uniform and geometric matrices yield similar curves.

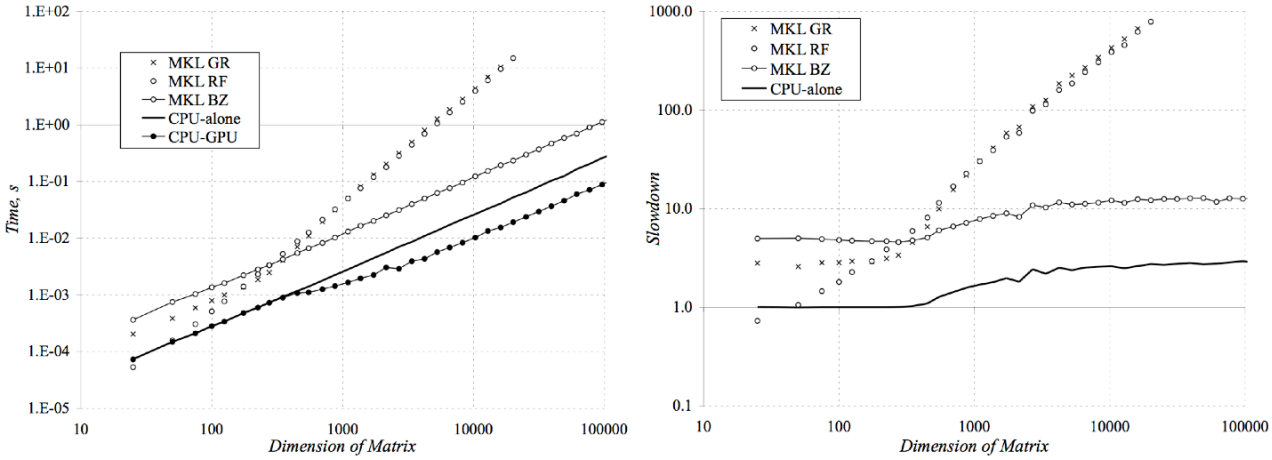

Figure 15: Glued matrices: runtimes of different implementations and their slowdowns relative to the CPU-GPU version. The benefits of using the GPU are small due to small inherent parallelism in the problem. Bisection algorithms run in linear time due to strongly clustered eigenvalues.