GPU acceleration of Newton's method for large systems of polynomial equations in double double and quad double arithmetic

Jan Verschelde* and Xiangcheng Yu†
Department of Mathematics, Statistics, and Computer Science
University of Illinois at Chicago, 851 South Morgan (M/C 249)
Chicago, IL 60607-7045, USA

February 11, 2014

Abstract

In order to compensate for the higher cost of double double and quad double arithmetic when solving large polynomial systems, we investigate the application of NVIDIA Tesla C2050, K20C, and K40 general purpose graphics processing units. As the dimension equals several thousands, the cost to compute one QR decomposition is sufficiently large so that the achieved speedups also compensate for the cost of data transfer of the evaluated polynomials and their derivatives. In cases where the polynomial evaluation and differentiation is relatively inexpensive, we could thus afford to leave this stage in the computation at the host.

Acceleration of our modified Gram-Schmidt method enables to solve linear systems in the least squares sense in higher dimension and with greater accuracy. In particular, with acceleration it takes less time to solve a system of dimension 4,096 in double double complex arithmetic than a system of dimension 2,048 in double complex arithmetic without acceleration. On a polynomial system where the cost of evaluation and differentiation grows linearly in the dimension with an accelerated Newton's method we obtain double digit speedups in double double complex arithmetic.

Key words and phrases. double double arithmetic, differentiation and evaluation, general purpose graphics processing unit (GPU), Newton's method, least squares, massively parallel algorithm, modified Gram-Schmidt method, polynomial system, QR decomposition, quad double arithmetic, quality up.

1 Introduction

We investigate the application of general purpose graphics processing units (GPUs) to solving large systems of polynomial equations with numerical methods. Because large systems not only lead to an increased number of operations, but also to more accumulation of numerical roundoff errors and therefore to the need to calculate in a precision that is higher than the common double precision. Motivated by the need of higher numerical precision, we can formulate our goal more precisely. With masssively parallel algorithms we aim to offset the extra cost of double double and quad double arithmetic [15, 23] and achieve quality up [2], a project we started in [31].

In this paper we report on solving the restrictions imposed on the dimensions of the problems our original implementation could solve. Our original massively parallel algorithms for evaluation and differentiation of polynomials [32] and for the modified Gram-Schmidt method [33] were written with a fine

*email: jan@math.uic.edu, URL: www.math.uic.edu/∼jan
†email: xyu30@uic.edu

granularity, making intensive use of the shared memory. The limitations on the capacity of the shared memory led to restrictions on the dimensions on the problems we could solve. These problems worsened for higher levels of precision, in contrast to the rising need for more precision in higher dimensions. While working in relatively low dimensions, we could not afford much data transfer between host and device, so that both the evaluation and differentiation stage and the solving on the linear system had to be executed on the device.

The revised versions of our massive parallel algorithms utilize the global memory of the device to store intermediate results between different kernel launches. Large vectors are chopped up in parts, read into shared memory for processing. Since the processing can happen in parallel by many more blocks of threads, the parallelism is increased, compensating for an increased access to global memory. For all levels of precision we can solve linear systems of several thousands of equations and variables. As the processing time for one QR decomposition in such large dimensions is already considerable, the evaluation and differentation stage can be separated from the solving of a linear system. The higher arithmetical cost of solving one linear system on the device compensates for the data transfer between the host and the device. This implies that the host could do the evaluation and differentiation of the polynomial system.

Problem Statement. Our problem is to accelerate Newton's method for large polynomial systems, aiming to offset the overhead cost of double double and quad double complex arithmetic. We do not make assumptions on the shape and structure of the polynomial systems. For accuracy and application to overdetermined systems, we solve linear systems in the least squares sense, based on a QR decomposition of the Jacobian matrix, which leads to the method of Gauss-Newton.

Related Work. As the QR decomposition is of fundamental importance in applied linear algebra many parallel implementations have been investigated by many authors, see e.g. [1], [3]. A high performance implementation of the QR algorithm on GPUs is described in [18]. In [6], the performance of CPU and GPU implementations of the Gram-Schmidt were compared. A multicore QR factorization is compared to a GPU implementation in [24]. GPU algorithms for approaches related to QR and Gram-Schmidt are for lattice basis reduction [4] and singular value decomposition [9].

The application of extended precision to BLAS is described in [22], see [8] for least squares solutions. The implementation of BLAS routines on GPUs in triple precision (double + single float) is discussed in [27]. In [28], double double arithmetic is described under the section of error-free transformations. An implementation of interval arithmetic on CUDA GPUs is presented in [7].

The other computationally intensive stage in the application of Newton's method is the evaluation and differentiation of the system. Parallel automatic differentiation techniques are described in [5], [12], and [29].

Concerning the GPU acceleration of polynomial systems solving, we mention two recent works. A subresultant method with a CUDA implementation of the FFT to solve systems of two variables is presented in [25]. In [20], a CUDA implementation for an NVIDIA GPU of a multidimensional bisection algorithm is discussed.

Our contributions. Our goal is to offset the cost of double double and quad double arithmetic in the context of numerical algorithms for solving polynomial systems. Although we have pushed the dimensions of the problems we consider into the thousands, we currently utilize only one CPU and one GPU. Because our computations are geared towards extended precision arithmetic which carry a higher cost per operation, we can afford a fine granularitiy in our parallel algorithms.

Our experiments show that GPU acceleration compensates for at least one level of precision: if we can afford double arithmetic on one single core of a CPU, then we can afford double double arithmetic with GPU acceleration. This result confirms that GPUs are well suited for well structured linear algebra problems. For polynomial evaluation and differentiation applied in Newton's method, for large enough dimensions, the cost of solving in the least squares sense is sufficiently high enough to compensate for data transfers so one could leave the polynomial evaluation and differentiation (which is often irregular) to a multicore implementation, as multithreading is more flexible than GPU acceleration.

Our results are described in the following four sections. First we explain our modifications to our accelerated Gram-Schmidt method so we can handle linear systems with dimensions over one thousand. Section three shows the computational results obtained when solving random complex linear systems in the least squares sense, in double, double double, and quad double arithmetic. We then apply our accelerated linear system solver in Newton's method on a problem arising from the discretization of an integral equation. In section five, we outline a new evaluation and differentation method to evaluate products of variables in factored form, to overcome the regularity assumptions imposed by shared memory limitations.

2 Orthogonalization and Delayed Normalization

Before describing our massively parallel algorithms for the modified Gram-Schmidt method [21], we formalize the notations. We typically compute with complex numbers and follow notations in [10] for the complex conjugated inner product $\mathbf{x}^H\mathbf{y}$. Pseudo code of the modified Gram-Schmidt orthogonalization method is listed in Figure 1.

Input: $A \in \mathbb{C}^{m \times n}$.
Output: $Q \in \mathbb{C}^{m \times n}$, $R \in \mathbb{C}^{n \times n}$: $Q^HQ = I$,
R is upper triangular, and $A = QR$.
let $\mathbf{a}_k$ be column k of A
for k from 1 to n do
  $r_{k,k} := \sqrt{\mathbf{a}_k^H \mathbf{a}_k}$
  $\mathbf{q}_k := \mathbf{a}_k / r_{k,k}$, $\mathbf{q}_k$ is column k of Q
  for j from $k+1$ to n do
    $r_{k,j} := \mathbf{q}_k^H \mathbf{a}_j$
    $\mathbf{a}_j := \mathbf{a}_j - r_{k,j}\mathbf{q}_k$

Figure 1: The modified Gram-Schmidt orthogonalization algorithm.

The modified Gram-Schmidt method computes the the QR decomposition of a matrix A, which allows to solve the linear system $A\mathbf{x} = \mathbf{b}$ in the least squares sense, minimizing $||\mathbf{b} - A\mathbf{x}||_2^2$. In the reduction of $A\mathbf{x} = \mathbf{b}$ to an upper triangular system $R\mathbf{x} = Q^H\mathbf{b}$, we do not compute $Q^H\mathbf{b}$ separately. As recommended in [16, §19.3] for numerical stability the modified Gram-Schmidt method is applied to the matrix A augmented with $\mathbf{b}$:

$$\left[\begin{array}{cc} A & \mathbf{b} \end{array}\right] = \left[\begin{array}{cc} Q & \mathbf{q}_{n+1} \end{array}\right] \left[\begin{array}{cc} R & \mathbf{y} \\ 0 & z \end{array}\right]. \tag{1}$$

Because $\mathbf{q}_{n+1}$ is orthogonal to the column space of Q: $||\mathbf{b} - A\mathbf{x}||_2^2 = ||R\mathbf{x} - \mathbf{y}||_2^2 + z^2$ and $\mathbf{y} = Q^H\mathbf{b}$.

The algorithm in Figure 1 starts with the computation of the complex conjugated inner product $\mathbf{a}_k^H\mathbf{a}_k$, followed by the normalization $\mathbf{q}_k := \mathbf{a}_k / r_{k,k}$, where $r_{k,k} := \sqrt{\mathbf{a}_k^H\mathbf{a}_k}$. For the inner product, we load the components of an m-dimensional vector into shared memory. Denoting the number of components that fit into the shared memory by K (its capacity), then let $L = \lceil m/K \rceil$ be the number of rounds it takes to compute

$$\mathbf{a}_k^H\mathbf{a}_k = \sum_{i=0}^{L-1} \sum_{j=0}^{K-1} \overline{a}_{k,i\star K+j} a_{k,i\star K+j}, \tag{2}$$

where the indexing of the components of a vector starts at zero and $\overline{a}$ denotes the complex conjugate of $a \in \mathbb{C}$. The value for K is typically a multiple of the warp size and equals the number of threads in a block.

In (2), the index j is the index of the thread in a block, so the inner loop is performed simultaneously in one step by all threads in the block. The outside loop on i is done in a sum reduction and takes $\log_2(L)$ steps. The computation of $\mathbf{a}_k^H \mathbf{a}_k$ for an n-dimensional vector $\mathbf{a}_k$ is reduced to m memory accesses, L steps to make all partial sums $\sum_{j=0}^{K-1} \overline{a}_{k,i\star K+j} a_{k,i\star K+j}$, and then $\log_2(L)$ steps for the outer sum.

For the reduction, we compute the inner product $r_{k,j} := \mathbf{q}_k^H \mathbf{a}_j$ of two m-vectors:

$$\begin{array}{ccc} \mathbf{q}_k & \mathbf{a}_j & \mathbf{q}_k^H \mathbf{a}_j \\ \left[\begin{array}{c} q_{k,0} \\ q_{k,1} \\ \vdots \\ q_{k,m-1} \end{array}\right] & \left[\begin{array}{c} a_{j,0} \\ a_{j,1} \\ \vdots \\ a_{j,m-1} \end{array}\right] & \left[\begin{array}{c} \bar{q}_{k,0} \star a_{j,0} \\ \bar{q}_{k,1} \star a_{j,1} \\ \vdots \\ \bar{q}_{k,K-1} \star a_{j,m-1} \end{array}\right] \end{array} \tag{3}$$

As we can keep K components of each vector in shared memory, thread t in a block computes $\bar{q}_{k,t} \star a_{j,t}$. If we may override $\mathbf{q}_k$, then $2m$ shared memory locations suffice, but we still need $\mathbf{q}_k$ for $\mathbf{a}_j := \mathbf{a}_j - r_{k,j}\mathbf{q}_k$. In total we need $3m$ shared memory locations to perform the reductions. Similar to the inner product for the norm of $\mathbf{a}_k$, the computation of $\mathbf{q}_k^H \mathbf{a}_j$ is performed in L rounds, where $L = \lceil 3m/K \rceil$, for the capacity K of shared memory.

The calculation of the inner products in L rounds is the first modification to our original massively parallel Gram-Schmidt implementation. The second modification is the delay of the normalization. In the next paragraph we explain the need for this delay.

In the reduction stage, the inner j-loop is executed by $n-k$ blocks of threads. Every block of threads performs the normalization of the k-th pivot column before proceeding to the reduction. The first block writes the normalized vector into global memory, all other blocks write the reduced vectors into global memory. In the new revised implementation, each vector is processed in several rounds and is read from global memory into shared memory not only at the beginning of the calculations. For large dimensions, not all blocks can be launched simultaneously. It may even be that the block that will reduce the last column is not even scheduled for launching at a time when the first block has finished its writing of the normalized $\mathbf{a}_k$ into global memory.

As some block would load in (partially) normalized vectors in the reduction stage, we propose to delay the normalization to the next iteration of the k-loop in the algorithm in Figure 1. At each iteration, the first block writes the norm of the current pivot column to a location in global memory and normalizes the previous pivot column, dividing every component of the previous pivot column by its norm stored in global memory and writes then the normalized previous column into global memory. With delayed normalization, the column $\mathbf{q}_k$ is computed last and is only stored in step $k+1$. At the very end of the algorithm, there is one extra kernel launch for the normalization that leads to $\mathbf{q}_n$.

The application of shared memory to reduce global memory traffic is referred to as tiling [19, pages 108-109]. Our tiles consist of slices of one column as we assign one column to one block. If we want to reduce the number of kernel launches, we could assign multiple (adjacent) columns to one block to make proper tiles as submatrices of the original matrix.

The third modification concerns the back substitution to compute the least squares solution to $Rx = Q^H\mathbf{b}$. Limited by the capacities of shared memory in our previous implementation only one block of threads performed the back substitution. For larger dimensions, denoting $Q^H\mathbf{b}$ by $\mathbf{y}$, the computation of

$$r_{\ell,\ell} x_\ell = y_\ell - \sum_{j=0}^{\ell-1} r_{\ell,j} x_j = y_\ell - \sum_{i=0}^{L-1} \sum_{j=0}^{K-1} r_{\ell,i\star K+j}, \tag{4}$$

where $L = \lceil m/K \rceil$, for the capacity K of shared memory. The main difference with our previous implementation is that now L blocks can work simultaneously at the evaluation of various components of (4).

The pivot block computes the actual components of the solution, while the other blocks compute the reductions for components at the low indices and write the reductions of the right hand side vector into global memory for processing in later stages. The first stage of the back substitution launches L blocks, the next stage launches $L-1$ blocks, followed by $L-2$ blocks in the third stage, etc. So there are as many stages as the value of L, each stage launching one fewer block as the previous stage.

3 Computational Results

In this section we describe results with our preliminary implementations. We first examine the potential for speedup with acceleration on matrices with random complex numbers. For dimensions as large as 1024, the solution of *one* linear system is sufficiently computationally intensive and the speedups are encouraging enough that we could run Newton's method on a benchmark problem where the cost of evaluation and differentiation is linear in the dimension. We obtained good speedups (the results are very good for double double complex arithmetic) when the host performs the evaluation and differentation and only the linear algebra is accelerated by the device, as demonstrated in the next section.

3.1 hardware and software

Our main target platform is the NVIDIA Tesla K20C, which has 2496 cores with a clock speed of 706 MHz, hosted by a Red Hat Enterprise Linux workstation of Microway, with Intel Xeon E5-2670 processors at 2.6 GHz. Our code was developed with version 4.4.7 of gcc and version 5.5 of the CUDA compiler.

Our other computer is an HP Z800 workstation with 3.47 GHz Intel Xeon X5690, running Red Hat Enterprise Linux, hosting the NVIDIA Tesla C2050 has 448 cores at a clock speed of 1147 Mhz. The same compilers where used on both computers.

We participated to the NVIDIA GPU Test Drive program of Microway and received access to a computer with two 10-core Xeon E5-2680v2 2.8GHz CPUs and one NVIDIA Tesla Atlas GPU, running CentOs Linux 6. The version of the gcc compiler is 4.4.6 and we used version 5.5 of the CUDA compiler.

The C++ code for the Gram-Schmidt method to run on the host is directly based on the pseudo code and we did not perform any optimizations. Our speedups might decrease once compared to the performance of the XBLAS [22] and in particular to [8]. Our C++ code served mainly to verify the correctness of our GPU code. The code is available at github in the directory src/GPU/MGS2 of PHCpack.

3.2 solving random complex linear systems

The host generates an n-by-$(n+1)$ matrix $[A\ \mathbf{b}]$ of complex random numbers, uniformly distributed on the unit circle, generating random angles $\theta \in [0, 2\pi)$ and taking the tuple $(\cos(\theta), \sin(\theta))$ in standard double precision as the real and imaginary parts of the complex numbers. In the serial runs, the host applies the modified Gram-Schmidt method to compute a QR decomposition, followed by a back substitution to solve the linear system defined by $A\mathbf{x} = \mathbf{b}$. In the accelerated runs, the data is sent to the device for the computation of the QR decomposition and the back substitution. On return, the matrices Q, R, and the solution $\mathbf{x}$ are sent back from the device to the host. The wall clock time (listed as real in the tables below) is used to compute the speedup of the accelerated runs.

Table 1 lists the timings (with real, user, and sys) obtained from the `time` command for the serial runs on one CPU core. Times are expressed in minutes (m) and seconds (s). As the cost grows cubic in the dimension n, observe that the time is multiplied by factor a bit over 8 when the dimension n is doubled. Going from double to double double arithmetic carries an overhead factor of about 10, while the overhead factor from double double to quad double arithmetic is closer to six. The higher than expected overhead

Table 1: Timings of the modified Gram-Schmidt method on a random n-by-$n+1$ complex matrix, followed by a back substitution done on one core of an Intel Xeon 2.60 Ghz processor in double (D), double double (DD), and quad double complex arithmetic (QD).

p	n	real	user	sys
D	1024	39.872s	39.649s	0.028s
	2048	5m23.402s	5m22.850s	0.075s
	3072	18m19.975s	18m18.073s	0.289s
	4096	43m18.139s	43m13.184s	0.328s
	5120	83m59.885s	83m50.417s	0.464s
DD	1024	7m 1.064s	7m 0.199s	0.055s
	2048	56m 2.627s	55m56.677s	0.586s
	3072	189m22.950s	189m 4.729s	0.575s
	4096	452m53.113s	452m12.340s	2.046s
QD	1024	41m 9.521s	41m 5.699s	0.092s
	2048	329m 4.188s	328m35.558s	0.346s

factor ten could be due to internal memory constraints caused by the doubling of the occupied memory when going from double to double double arithmetic. Figure 2 visualizes the data of Table 1.

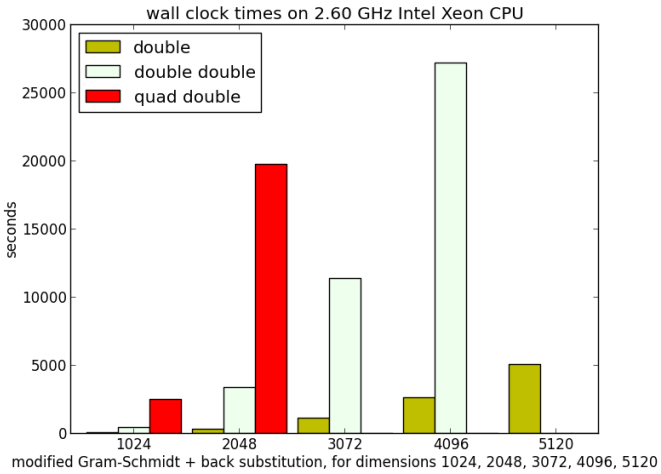

Figure 2: Figure visualizing the data of Table 1. The plot shows that the following times are of the same magnitude at 1024 in QD (41 min), 2048 in DD (56 min), and 4096 in DD (43 min).

Table 2 lists the accelerated times on the K20C for various block sizes (BS), the number of threads in a block. As each streaming multiprocessor of the K20C has 192 cores, we observe an increase of performance as the value for BS increases. The largest value for BS we could afford was 256, due to limitations of the capacity of shared memory. Lower limits on the value for BS are determined by the maximal number of rounds (the value for L in the previous section) which equals 32. So for 2048, we start our experiments with BS equal to 64, as $2048 = 32 \times 64$. The speedup are computed dividing the corresponding real times of Table 1 by the values for real obtained by acceleration. Already on the solving of *one* linear system (which includes times for the data transfers between host and device) we reach speedups higher than 20 for large enough values of BS. Figure 3 visualizes the data of Table 2.

Table 2: Accelerating the Gram-Schmidt method by the K20C on a random complex n-by-$(n+1)$ matrix followed by back substitution in double complex arithmetic. The block size BS (the number of threads in one block) is increased by 32 up to the shared memory limit.

n	BS	real	user	sys	speedup
1024	32	5.067s	3.138s	1.501s	7.87
1024	64	3.045s	1.747s	1.099s	13.09
1024	96	2.544s	1.434s	0.966s	15.67
1024	128	2.159s	1.116s	0.801s	18.47
1024	160	2.140s	1.083s	0.859s	18.63
1024	192	1.971s	1.039s	0.782s	20.23
1024	224	1.862s	0.874s	0.794s	21.41
1024	256	1.712s	0.807s	0.744s	23.29
2048	64	18.112s	10.892s	6.950s	17.86
2048	96	14.568s	8.866s	5.516s	22.20
2048	128	11.414s	6.841s	4.342s	28.33
2048	160	10.610s	6.335s	4.042s	30.48
2048	192	9.595s	5.717s	3.634s	33.71
2048	224	9.244s	5.530s	3.461s	34.99
2048	256	8.098s	4.768s	3.147s	39.94
3072	96	44.692s	26.928s	17.518s	24.61
3072	128	35.435s	21.003s	14.198s	31.04
3072	160	33.395s	19.848s	13.251s	32.94
3072	192	28.310s	17.109s	11.002s	38.85
3072	224	26.342s	15.780s	10.390s	41.76
3072	256	24.411s	14.616s	9.542s	45.06
4096	128	1m20.761s	47.926s	32.490s	32.17
4096	160	1m14.422s	44.555s	29.596s	34.91
4096	192	1m 6.437s	43.248s	22.877s	39.11
4096	224	1m 1.094s	39.699s	21.001s	42.53
4096	256	55.140s	32.937s	21.898s	47.12
5120	160	2m20.774s	1m20.936s	59.490s	35.80
5120	192	2m 4.366s	1m13.106s	50.782s	40.52
5120	224	1m53.530s	1m 7.168s	45.975s	44.39
5120	256	1m45.579s	1m 3.446s	41.842s	47.74

Table 3 lists timings for acceleration in double double and quad double arithmetic. For double double arithmetic, the speedups are really great, most likely through to increased benefits of the access to global memory which goes faster on the device than on the host. Similar as in double arithmetic, we notice benefits from increasing the block size, although now the upper limits are for double double and quad double arithmetic are respectively 128 and 64, due to constraints on the shared memory. Speedups are the ratios between the wall clock times of Table 1 and the wall clock times of Table 3. The largest speedup of 131.24 on a problem of dimension 3,072 in double double complex arithmetic reduces the time of more than 3 hours (189m22.950s) down to less than 1.5 minutes (1m25s). For the largest dimension we can do in double double complex arithmetic we do not have unaccelerated times. Speedups on solving *one* linear system with quad double complex arithmetic are not so good, because the data transfer between host and device is substantially larger. Figure 4 visualizes Table 3.

We end this section with a comparison with two other GPUs: the older C2050 and the newer K40. Table 4 displays timings for the C2050. Although the CPU frequency of the host for the C2050 was scaled

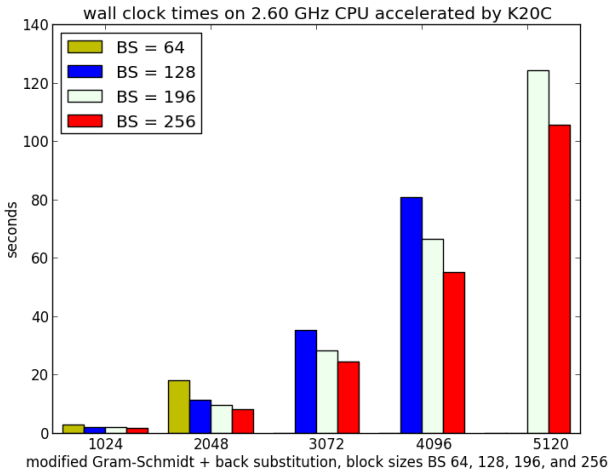

Figure 3: Figure visualizing the data of Table 2. The plot shows that larger block sizes are more beneficial for larger dimensions.

to a fixed 2.66GHz (to be more in line with the host of the K20C), the speedups are based on the sys time. Similar to the K20C, we observe improved performance for increased values of the block size (number of threads in one block), but as the number of cores per streaming multiprocessor of the C2050 is only 32 and not 192 as on the K20C, we observe an increase in times as the block size goes past 128. That the speedups for double double complex arithmetic are less than 1.0 is odd. Perhaps it could be that the higher clock speed of the C2050 (1147MHz) versus that of the K20C (706MHz) may play an important role for matrices of size 1,024.

We conducted a comparison with the new NVIDIA K40 runs, with timings displayed in Table 5. As the clock speed of the host for the K40 is slightly higher (2.80Ghz instead of 2.60Ghz) than the host for the K20C, the system time is used to measure the speedup of the K40 over the K20C. Unlike the oddity observed with the C2050, the speedups obtained by the K40 over the K20C are consistent with the theoretical peak performace of the K40.

Although the host of the K40 has a slightly higher clock speed (2.80GHz) than the host of the K20C (2.60GHz), we recompute the highest speedup in Table 3, for dimension 4,096 in double double complex arithmetic. The original time of 452m53.113s (more than 7.5 hours) is reduced to 2m39.031s, giving a speedup factor of 170.87. As for correctness in solving large dimensional problems double double precision may be the default precision, observe that solving a linear system in the least squares sense without GPU acceleration already takes 7 minutes (see Table 1), in dimension 1,024, which is far less than the accelerated times for dimension 4,096. With GPU acceleration we can solve much larger linear systems in double double complex arithmetic.

4 Newton's method

Given a system $f(\mathbf{x}) = \mathbf{0}$, with $\mathbf{x} = (x_1, x_2, \ldots, x_n)$, we denote the matrix of all partial derivatives of f as J_f. Given an initial approximation $\mathbf{x}_0$ for a solution of $f(\mathbf{x}) = \mathbf{0}$, the application of one step in Newton's method happens in two stages:

1. Evaluate J_f and f at $\mathbf{x}_0$: $A = J_f(\mathbf{x}_0)$ and $\mathbf{b} = -f(\mathbf{x}_0)$.

Table 3: Accelerating the Gram-Schmidt method by the K20C on an n-by-$(n+1)$ matrix, followed by a back substitution, in double double (DD) and quad double complex arithmetic (QD) for various block sizes BS.

p	n	BS	real	user	sys	speedup
DD	1024	32	11.846s	8.563s	3.050s	35.54
	1024	64	6.608s	4.437s	1.967s	63.72
	1024	96	5.359s	3.440s	1.665s	78.57
	1024	128	4.270s	2.749s	1.320s	98.61
	2048	64	44.798s	27.813s	16.720	75.06
	2048	96	34.929s	21.765s	12.860s	96.27
	2048	128	27.039s	16.838s	9.922s	124.36
	3072	96	1m49.947s	1m 6.248s	43.375s	103.35
	3072	128	1m26.581s	51.433s	34.724s	131.24
	4096	128	3m21.074s	1m57.756s	1m22.789s	135.14
QD	1024	32	4m12.664s	3m22.920s	49.078s	9.77
	1024	64	2m30.463s	1m53.848s	36.221s	16.41
	2048	64	19m20.893s	11m48.509s	7m30.749s	17.01

Table 4: Accelerating the Gram-Schmidt method by the C2050, on an 1024-by-1024 matrix, followed by a back substitution, in double (D), double double (DD) and quad double (QD) complex arithmetic for various block sizes BS. Speedups are computing dividing the sys times in this table by those corresponding sys times from Table 2 and Table 3.

p	BS	real	user	sys	speedup
D	32	5.012s	2.654s	1.861s	1.24
	64	3.494s	1.533s	1.451s	1.32
	96	3.154s	1.501s	1.596s	1.65
	128	2.902s	1.072s	1.247s	1.56
	160	3.125s	1.229s	1.236s	1.44
	192	3.126s	1.450s	1.427s	1.82
	224	3.184s	1.534s	1.594s	2.01
	256	3.029s	1.430s	1.542s	2.07
DD	32	10.866s	7.260s	3.468s	0.61
	64	6.834s	4.042s	2.339s	0.74
	96	6.259s	3.421s	2.128s	0.96
	128	5.880s	3.543s	2.218s	0.95
QD	32	3m47.729s	2m58.814s	48.703s	0.99
	64	2m34.034s	1m50.693s	41.865s	1.16

2. Solve the linear system $A\Delta\mathbf{x} = \mathbf{b}$ and update $\mathbf{x}_0$ to $\mathbf{x}_1 := \mathbf{x}_0 + \Delta\mathbf{x}$.

Stating the stages explicitly as above we emphasize the separation between the two stages in solving general polynomial systems where the shape and structure of the polynomials varies widely between almost linear to sparse systems with high degree monomials, see for example the benchmark collection of PHCpack [30].

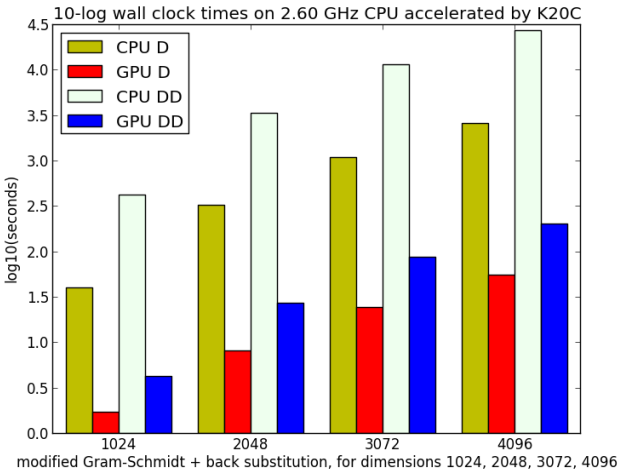

Figure 4: Wall clock times at *logarithmic scale* from Table 1, Table 2, and Table 3. Observe that the rightmost bar is shorter than fifth bar from the left: solving a linear system of dimension 4,096 with the GPU takes less time than solving a linear system of dimension 2,048 on the CPU with double complex arithmetic.

Table 5: Accelerating the Gram-Schmidt method by the K40, on an n-by-$(n+1)$ matrix, followed by a back substitution, in double complex arithmetic (DD), block size 256, in double double complex arithmetic (DD), block size 128, and quad double complex arithmetic (QD), block size 64. Speedups are computed dividing the sys times of Table 2 and Table 3 by the sys times on the K40.

p	n	real	user	sys	speedup
D	1024	1.303s	0.697s	0.496s	1.5
	2048	6.673s	4.136s	2.428s	1.23
	3072	20.342s	12.577s	7.628s	1.25
	4096	46.284s	28.287s	17.823s	1.23
	5120	1m29.107s	54.447s	34.407s	1.22
DD	1024	3.264s	2.206s	0.950s	1.39
	2048	21.214s	13.458s	7.650s	1.30
	3072	1m 8.450s	41.942s	26.285s	1.32
	4096	2m39.031s	1m37.148s	1m 1.397s	1.35
QD	1024	2m 0.069s	1m32.061s	27.672s	1.31
	2048	15m24.871s	9m44.882s	5m37.872s	1.33

4.1 The Chandrasekhar H-Equation

The system arises from the discretization of an integral equation. The problem was treated with Newton's method in [17] and added to a collection of benchmark problems in [26]. In [11], the system was studied

with methods in computer algebra. We follow the formulation in [11]:

$$
\begin{aligned}
& f_i(H_1, H_2, \ldots, H_n) \\
& \quad = 2nH_i - cH_i \left(\sum_{j=0}^{n-1} \frac{i}{i+j} H_j \right) - 2n = 0, \\
& i = 1, 2, \ldots, n,
\end{aligned} \tag{5}
$$

where c is some real nonzero constant, $0 < c \leq 1$. As we can write the equations for any dimension n, observe that the cost of evaluating the polynomials remains linear in n. Also the cost of evaluating the columns of the Jacobian matrix linear in n as only the diagonal elements contain n linear terms. The off-diagonal elements of the Jacobian matrix consists of at most one linear term. As the evaluation and differentiation cost for this problem is linear in n, this implies that the cost of one iteration of Newton's method is dominated by the cost for solving the linear system, which is cubic in n.

Although the total number of solutions grows as 2^n, there is always one real solution with all its components positive and relatively close to 1. Starting at $H_i = 1$ for all i leads to a quadratically convergent Newton's method. The value for the parameter c we used in our experiments is 33/64. As all coefficients in the system and the solution are real, the complex arithmetic is superfluous. Nevertheless, we expect the speedups to be the same if we would use only real arithmetic.

Although in our methodology, not taking advantage of the shape and structure of the polynomial system, it does not seem possible to obtain correct results without the use of double double arithmetic, it may very well be that the Jacobi matrix at the interesting solution is diagonally dominant and that iterative methods in double arithmetic will do very well to solve this particular benchmark problem.

4.2 computational results

To run Newton's method on this system, the experimental code is displayed in Figure 5.

for a number of iterations :

1. The host evaluates and differentiates the system
   at the current approximation.
   This result of the evaluation and differentiation
   is stored in an n-by-$(n+1)$ matrix $[A\ \mathbf{b}]$,
   with $\mathbf{b} = -f(H_1, H_2, \ldots, H_n)$.
   The first component of $\mathbf{b}$ is printed.
2. $A\Delta\mathbf{x} = \mathbf{b}$ is solved in the least squares sense,
   either entirely by the host; or
   if accelerated, then
   - 2.1 the matrix $[A\ \mathbf{b}]$ is transferred
     from the host to the device;
   - 2.2 the device does a QR decomposition on $[A\ \mathbf{b}]$
     and back substitution on the system $R\Delta\mathbf{x} = \mathbf{y}$;
   - 2.3 the matrices Q, R, and the solution $\Delta\mathbf{x}$
     are transferred from the device to the host.
3. The host performs the update $\mathbf{x} = \mathbf{x} + \Delta\mathbf{x}$
   to compute the new approximation.
   The first component of $\Delta\mathbf{x}$ and $\mathbf{x}$ are printed.

Figure 5: Experimental setup to accelerate Newton's method.

In the runs, we observed that taking larger block sizes led to less accurate results, even to the extent where runs in double arithmetic with dimension 1,024 did not lead to correct results. Even if the QR decomposition may still be computed correctly, the back substitution stage incurs a large propagation of round off, the first component of the solution is subject to all errors of the half a million computed numbers in R. To be precise, for $n = 1,024$, there are 524,800 $(512 \times (1024 + 1))$ floating points numbers in R. For the GPU acceleration, taking smaller block sizes results in taking subsums of sequences of smaller size which leads to less roundoff propagation.

Table 6 lists results for runs in double double and quad double complex arithmetic when Newton's method is done *entirely* by the device. For dimension 1,024, first only the linear system solving was accelerated, which led to wall clock times of 41.193s in double double complex arithmetic and 15m34.527s in quad double complex arithmetic, leading to respective speedups of 62.18 and 16.30, matching very closely the speedups of 63.72 and 16.41 in Table 3. Performing the entire Newton's method on the device did not do much for quad double complex arithmetic, but it did improve the speedups for double double complex arithmetic: from 62.18 to 72.72 (for block size 64) and to 127.42 (for block size 128). When running multiple stages of Newton's method on the device, the speedups improve: 124.36 for solving one linear system in dimension 2,048 in double double complex arithmetic becomes 136.92 when running six iterations.

In Table 6, observe the quality up: it takes less than twice as long to run the accelerated Newton's method in quad double complex arithmetic than the unaccelerated run in double double complex arithmetic.

Figure 6 visualizes the data in Table 6.

Table 6: Running six iterations of Newton's method in complex double double arithmetic (DD) and quad double arithmetic (QD), on one core of the host (CPU) and with GPU acceleration (GPU), with block size equal to 128 (DD) and 64 (QD).

p	n	real	user	sys	speedup
DD	CPU 1024	42m41.480s	42m37.692s	0.038s	
	GPU 1024	20.102s	11.664s	8.236s	127.42
	CPU 2048	341m47.998s	341m18.009s	0.362s	
	GPU 2048	2m29.770s	1m26.373s	1m03.014s	136.92
QD	CPU 1024	253m51.126s	253m24.170s	4.802s	
	GPU 1024	15m11.362s	9m28.399s	5m41.532s	16.71
	CPU 2048	2027m40.726s	2024m38.715s	3.055s	
	GPU 2048	110m51.042s	63m21.470s	47m21.105s	18.29

5 Factored Evaluation and Differentiation

In our preliminary implementation in [32] to evaluation and differentiate systems of multivariate polynomials we imposed regularity assumptions. In particular, we expected a fixed number of variables with nonzero exponent in each monomial. Our method considered the evaluation and differentiation of one product of variables (the so-called Speelpenning example [14], although it should be attributed to Arthur Sedgwick [13]) as one job to be executed in its entirety by each thread. In this section we describe a method to remove this regularity assumption.

The main idea is to split a product of variables into factors. The size of each factor is such that all variables participating in the factor fit into shared memory. In the example below we consider a product of 16 variables and assume that the memory shared between all threads in a block equals 4, the size of each factor. Separate threads evaluate and differentiate products of variables:

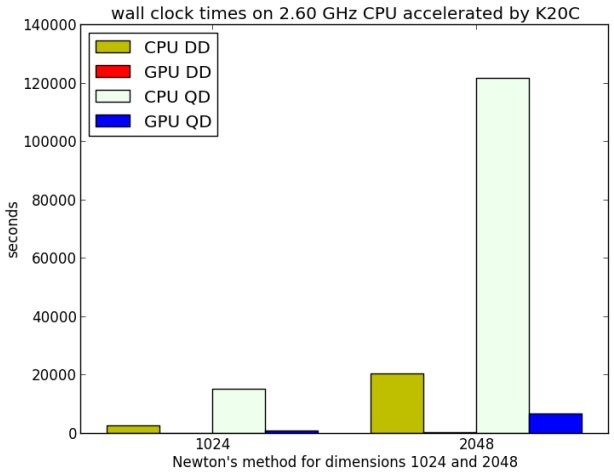

Figure 6: Figure visualizing the data of Table 6. Observe the height of the rightmost bar. Accelerating Newton's method in quad double complex arithmetic in dimension 2048 takes less than running Newton's method without acceleration with double double arithmetic in the same dimension.

$$\begin{array}{cccc} \overbrace{x_0x_1x_2x_3}^{f_0} & \overbrace{x_4x_5x_6x_7}^{f_1} & \overbrace{x_8x_9x_Ax_B}^{f_2} & \overbrace{x_Cx_Dx_Ex_F}^{f_3} \\ x_1x_2x_3 & x_5x_6x_7 & x_9x_Ax_B & x_Dx_Ex_F \\ x_0x_2x_3 & x_4x_6x_7 & x_8x_Ax_B & x_Cx_Ex_F \\ x_0x_1x_3 & x_4x_5x_7 & x_8x_9x_B & x_Cx_Dx_F \\ x_0x_1x_2 & x_4x_5x_6 & x_8x_9x_A & x_Cx_Dx_E \end{array} \quad (6)$$

Because a variable occurs in at most one factor, e.g.: $\dfrac{\partial}{\partial x_0}(f_0f_1f_2f_3) = \dfrac{\partial f_0}{\partial x_0}f_1f_2f_3$, the product rule for derivatives does not apply and all threads in one block need to be concerned only with the values of the variables that fit into their shared memory. Therefore, after the calculation of all derivatives of one factor, the derivatives of the product of all factors need to be multiplied with the products of the other factors in the second stage.

All derivatives in (6) of f_0, f_1, f_2, and f_3 are multiplied by respectively by $f_1f_2f_3$, $f_0f_2f_3$, $f_0f_1f_3$, and $f_0f_1f_2$. So we run the same algorithm as in (6) with variables $x_0 = f_0$, $x_1 = f_1$, $x_2 = f_2$, and $x_3 = f_3$.

The algorithm to multiply each derivative of a factor with all other factors follows the same lines as the original evaluation and differentiation of any product of variables. As the product of variables increase in size, we run multiple recursive applications of the evaluation and differentiation of factors of the product. The extra operations to combine the evaluated factors and the extra memory locations to hold the intermediate results is directly linear in the number of factors.

6 Conclusions and Outlook

For polynomial systems of dimensions of 1,024 and higher, GPU acceleration reduces the time it takes to solve a linear system in double double complex arithmetic from minutes to seconds and from hours to minutes. On an example we showed that, if the cost of evaluation and differentiation grows linearly in

the dimension, accelerating Newton's method in double double complex arithmetic still gives significant speedups, even if we apply the GPU acceleration only to the linear system solving. Polynomial systems of higher complexity we can either evaluate and differentiation on multiple cores on the host, or also move this stage to the device.

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant No. 1115777. The Microway workstation with the NVIDIA Tesla K20C was purchased through a UIC LAS Science award. We gratefully acknowledge Microway for providing access to a Tesla-accelerated compute cluster. In particular, the comparison between the K20C and the K40 were performed on Microway's Tesla GPU accelerated compute cluster.

References

[1] E. Agullo, C. Augonnet, J. Dongarra, M. Faverge, H. Ltaief, S. Thibault, and S. Tomov. QR factorization on a multicore node enhanced with multiple GPU accelerators. In *Proceedings of the 2011 IEEE International Parallel Distributed Processing Symposium (IPDPS 2011)*, pages 932–943. IEEE Computer Society, 2011.

[2] S.G. Akl. Superlinear performance in real-time parallel computation. *The Journal of Supercomputing*, 29(1):89–111, 2004.

[3] M. Anderson, G. Ballard, J. Demmel, and K. Keutzer. Communication-avoiding QR decomposition for GPUs. In *Proceedings of the 2011 IEEE International Parallel Distributed Processing Symposium (IPDPS 2011)*, pages 48–58. IEEE Computer Society, 2011.

[4] T. Bartkewitz and T. Güneysu. Full lattice basis reduction on graphics cards. In F. Armknecht and S. Lucks, editors, *WEWoRC'11 Proceedings of the 4th Western European conference on Research in Cryptology*, volume 7242 of *Lecture Notes in Computer Science*, pages 30–44. Springer-Verlag, 2012.

[5] C. Bischof, N. Guertler, A. Kowartz, and A. Walther. Parallel reverse mode automatic differentiation for OpenMP programs with ADOL-C. In C. Bischof, H.M. Bücker, P. Hovland, U. Naumann, and J. Utke, editors, *Advances in Automatic Differentiation*, pages 163–173. Springer-Verlag, 2008.

[6] T. Brandes, A. Arnold, T. Soddemann, and D. Reith. CPU vs. GPU - performance comparison for the Gram-Schmidt algorithm. *The European Physical Journal Special Topics*, 210(1):73–88, 2012.

[7] S. Collange, M. Daumas, and D. Defour. Interval arithmetic in CUDA. In Wen mei W. Hwu, editor, *GPU Computing Gems Jade Edition*, pages 99–107. Elsevier, 2012.

[8] J. Demmel, Y. Hida, X.S. Li, and E.J. Riedy. Extra-precise iterative refinement for overdetermined least squares problems. *ACM Trans. Math. Softw.*, 35(4):28:1–28:32, 2009.

[9] B Foster, S. Mahadevan, and R. Wang. A GPU-based approximate SVD algorithm. In *Parallel Processing and Applied Mathematics*, volume 7203 of *Lecture Notes in Computer Science Volume*, pages 569–578. Springer-Verlag, 2012.

[10] G.H. Golub and C.F. Van Loan. *Matrix Computations*. The Johns Hopkins University Press, third edition, 1996.

[11] L. Gonzalez-Vega. Some examples of problem solving by using the symbolic viewpoint when dealing with polynomial systems of equations. In J. Fleischer, J. Grabmeier, F.W. Hehl, and W. Küchlin, editors, *Computer Algebra in Science and Engineering*, pages 102–116. World Scientific, 1995.

[12] M. Grabner, T. Pock, T. Gross, and B. Kainz. Automatic differentiation for GPU-accelerated 2D/3D registration. In C. Bischof, H.M. Bücker, P. Hovland, U. Naumann, and J. Utke, editors, *Advances in Automatic Differentiation*, pages 259–269. Springer-Verlag, 2008.

[13] A. Griewank. From the product example to pde adjoints, algorithmic differentiation and its application (invited talk). In V.P. Gerdt, W. Koepf, E.W. Mayr, and E.V. Vorozhtsov, editors, *Computer Algebra in Scientific Computing, 15th International Workshop, CASC 2013, Berlin, Germany*, volume 8136 of *Lecture Notes in Computer Science*, pages 130–135, 2013.

[14] A. Griewank and A. Walther. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*. SIAM, second edition, 2008.

[15] Y. Hida, X.S. Li, and D.H. Bailey. Algorithms for quad-double precision floating point arithmetic. In *15th IEEE Symposium on Computer Arithmetic (Arith-15 2001), 11-17 June 2001, Vail, CO, USA*, pages 155–162. IEEE Computer Society, 2001. Shortened version of Technical Report LBNL-46996, software at `http://crd.lbl.gov/~dhbailey/ mpdist/qd-2.3.9.tar.gz`.

[16] N.J. Higham. *Accuracy and Stability of Numerical Algorithms*. SIAM, 1996.

[17] C.T. Kelley. Solution of the Chandrasekhar h-equation by Newton's method. *J. Math. Phys.*, 21(7):1625–1628, 1980.

[18] A. Kerr, D. Campbell, and M. Richards. QR decomposition on GPUs. In D. Kaeli and M. Leeser, editors, *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units (GPGPU'09)*, pages 71–78. ACM, 2009.

[19] D.B. Kirk and W.W. Hwu. *Programming Massively Parallel Processors. A Hands-on Approach*. Morgan Kaufmann/Elsevier, second edition, 2013.

[20] R.A. Klopotek and J. Porter-Sobieraj. Solving systems of polynomial equations on a GPU. In M. Ganzha, L. Maciaszek, and M. Paprzycki, editors, *Preprints of the Federated Conference on Computer Science and Information Systems, September 9-12, 2012, Wroclaw, Poland*, pages 567–572, 2012.

[21] S.J. Leon, A. Björck, and W. Gander. Gram-Schmidt orthogonalization: 100 years and more. *Numerical Linear Algebra with Applications*, 20(3):492–532, 2013.

[22] X. Li, J. Demmel, D. Bailey, G. Henry, Y. Hida, J. Iskandar, W. Kahan, S. Kang, A. Kapur, M. Martin, B. Thompson, T. Tung, and D. Yoo. Design, implementation and testing of extended and mixed precision BLAS. *ACM Trans. Math. Softw.*, 28(2):152–205, 2002. This is a shortened version of Technical Report LBNL-45991.

[23] M. Lu, B. He, and Q. Luo. Supporting extended precision on graphics processors. In A. Ailamaki and P.A. Boncz, editors, *Proceedings of the Sixth International Workshop on Data Management on New Hardware (DaMoN 2010), June 7, 2010, Indianapolis, Indiana*, pages 19–26, 2010. Software at `http://code.google.com/p/gpuprec/`.

[24] Y. Luo and R. Duraiswami. Efficient parallel nonnegative least squares on multicore architectures. *SIAM J. Sci. Comp.*, 33(5):2848–2863, 2011.

[25] M.M. Maza and W. Pan. Solving bivariate polynomial systems on a GPU. *ACM Communications in Computer Algebra*, 45(2):127–128, 2011.

[26] J.J. Moré. A collection of nonlinear model problems. In *Computational Solution of Nonlinear Systems of Equations*, volume 26 of *Lectures in Applied Mathematics*, pages 723–762. AMS, 1990.

[27] D. Mukunoki and D. Takashashi. Implementation and evaluation of triple precision BLAS subroutines on GPUs. In *Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops. 21-25 May 2012, Shanghai China*, pages 1372–1380. IEEE Computer Society, 2012.

[28] S.M. Rump. Verification methods: Rigorous results using floating-point arithmetic. *Acta Numerica*, 19:287449, 2010.

[29] J. Utke, L. Hascoët, P. Heimbach, C. Hill, P. Hovland, and U. Naumann. Toward ajoinable mpi. In *Proceedings of the 10th IEEE International Workshop on Parallel and Distributed Scientific and Engineering Computing (PDSEC 2009)*, pages 1–8, 2009.

[30] J. Verschelde. Algorithm 795: PHCpack: A general-purpose solver for polynomial systems by homotopy continuation. *ACM Trans. Math. Softw.*, 25(2):251–276, 1999.

[31] J. Verschelde and G. Yoffe. Polynomial homotopies on multicore workstations. In M.M. Maza and J.-L. Roch, editors, *Proceedings of the 4th International Workshop on Parallel Symbolic Computation (PASCO 2010), July 21-23 2010, Grenoble, France*, pages 131–140. ACM, 2010.

[32] J. Verschelde and G. Yoffe. Evaluating polynomials in several variables and their derivatives on a GPU computing processor. In *Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops (PDSEC 2012)*, pages 1391–1399. IEEE Computer Society, 2012.

[33] J. Verschelde and G. Yoffe. Orthogonalization on a general purpose graphics processing unit with double double and quad double arithmetic. In *Proceedings of the 2013 IEEE 27th International Parallel and Distributed Processing Symposium Workshops (PDSEC 2013)*, pages 1373–1380. IEEE Computer Society, 2013.