manuscript No.
(will be inserted by the editor)

GPU Accelerated Greedy Algorithms for Compressed Sensing

Jeffrey D. Blanchard · Jared Tanner

Draft: 03 June 2012 / Accepted: date

Abstract For appropriate matrix ensembles, greedy algorithms have proven to be an efficient means of solving the combinatorial optimization problem associated with compressed sensing. This paper describes an implementation for graphics processing units (GPU) of hard thresholding, iterative hard thresholding, normalized iterative hard thresholding, hard thresholding pursuit, and a two-stage thresholding algorithm based on compressive sampling matching pursuit and subspace pursuit. The GPU acceleration of the former bottleneck, namely the matrix-vector multiplications, transfers a significant portion of the computational burden to the identification of the support set. The software solves high-dimensional problems in fractions of second which permits large-scale testing at dimensions currently unavailable in the literature. The GPU implementations exhibit up to 70x acceleration over standard Matlab central processing unit implementations using automatic multi-threading.

Keywords Combinatorial optimization · Compressed sensing · Sparse approximation · Greedy algorithms · Graphics processing units · Parallel computing · IHT · NIHT · HTP · CoSaMP · Subspace Pursuit

---

JDB was a National Science Foundation International Research Fellow at the University of Edinburgh under award NSF OISE 0854991 and is supported by NSF DMS 1112612. JT was supported by the Leverhulm Trust and an Nvidia Professor Partnership.

J.D. Blanchard
Department of Mathematics and Statistics, Grinnell College, Grinnell, IA 50112
Tel.: +1 641.269.3304
Fax: +1 641.269.4984
E-mail: jeff@math.grinnell.edu

J. Tanner
School of Mathematics, Univeristy of Edinburgh, Edinburgh, UK EH9 3JX
Tel.: +44 131.650.5057
Fax: +44 131.650.6553
E-mail: jared.tanner@ed.ac.uk

1 Introduction

A recent sub-discipline of sparse approximation is the theory of compressed sensing which gives rise to a particular combinatorial optimization problem. In the past several years many algorithms have been introduced with provable guarantees of solving this combinatorial optimization problem as a linear programming problem or with relatively efficient polynomial time algorithms. In particular, greedy algorithms have proven to be effective and efficient algorithms for solving the optimization problem in appropriate settings. These iterative algorithms are ripe for parallelization and implementation for a graphics processing unit (GPU).

This paper describes a GPU implementation of several greedy algorithms using three different GPU matrix-vector multiplications. In the remainder of this introduction, we introduce the combinatorial optimization problem inherent in compressed sensing and discuss some of the challenges in the GPU implementation. In Sec. 2 we describe the GPU implementation in detail. Section 3 outlines the use of the software, *GPU Accelerated Greedy Algorithms* for Compressed Sensing [5]. We present a performance evaluation of the software in the form of least squares fits to average timing values and comparisons to a serial CPU implementation in Sec. 4.

1.1 Combinatorial Optimization in Compressed Sensing

Compressed sensing [9, 15] is the study of reconstructing a sparse signal from a reduced number of linear measurements. A signal x is *k-sparse* if it has no more than k nonzero elements; we say the signal is *approximately* k-sparse if it has only k significant coefficients. Traditionally, a signal of length n is measured directly, $y = Ix = x$ where I is an $n \times n$ identity matrix. Under the assumption that a signal is k-sparse, or approximately k-sparse, the number of measurements required to capture all the pertinent information in the signal can be significantly reduced. In fact, the number of measurements, m, can be asymptotically proportional to the sparsity k, i.e. for fixed ratios $\delta, \rho \in (0,1)$, $(\frac{m}{n}, \frac{k}{m}) \to (\delta, \rho)$ as $(k,m,n) \to \infty$. More formally, a k-sparse signal $x \in \mathbb{R}^n$ is observed through inner products with the m rows of a measurement matrix $A \in \mathbb{R}^{m\times n}$ generating the measurements $y = Ax \in \mathbb{R}^m$. Having access only to y and the measurement process A, one seeks to recover the sparsest signal $\hat{x}$ which would generate these measurements. This leads to the combinatorial optimization problem

$$\hat{x} = \arg\min_{z\in\mathbb{R}^n} \|z\|_0 \text{ subject to } y = Az, \tag{1}$$

where $\|z\|_0$ counts the number of non-zero entries in z. In general, this combinatorial optimization problem is NP hard [29] and significant work over the past several years has focused on determining tractable algorithms which will produce the same solution to (1) for various classes of measurement matrices, A.

A large body of work has developed which, under the right conditions, establishes the convex relaxation of this problem as a suitable alternative, namely

$$\hat{x} = \arg\min_{z\in\mathbb{R}^n} \|z\|_1 \text{ subject to } y = Az, \tag{2}$$

where $\|z\|_1 = \sum_{i=1}^{n} |z_i|$ is the standard ℓ_1 norm. In particular, necessary and sufficient conditions on the measurement matrix A are known [10, 11, 14, 16–18] (among others) which ensure that the solutions to (1) and (2) coincide. The computational cost of obtaining the solution to (2) is clearly dependent on the algorithm used to obtain the solution. As (2) can be formulated as a linear programming problem, standard linear programming software can be employed. However, this problem has inherent structure, namely the sparsity of the signal x, and several algorithms have appeared over the past several years to exploit this structure [4, 12, 19–21, 24, 34, 35]. The complexity of some of these algorithms can limit their usefulness when the problem dimensions (k,m,n) grow to dimensions similar to real world applications.

A family of low-complexity greedy algorithms, in particular the family of thresholding algorithms, have been brought to bear on this problem. When the measurement matrix A meets certain sufficient conditions, it is now well-known that these greedy algorithms can be guaranteed to acquire the solution to (1). The simplest greedy algorithm is Hard Thresholding (HT) which makes an initial guess at the support of x and then projects the measurements y onto this support. The complexity of this algorithm is enviable, but the situations in which it is applicable require a large number of measurements, $m \sim k^2$. The early greedy algorithms studied in this setting are Matching Pursuit (MP) [26] and Orthogonal Matching Pursuit (OMP) [32, 33]. The low complexity of a single iteration of these algorithms makes them good candidates for solving (1) for low-dimensional problems, but since the number of iterations required to converge to a k-sparse solution is exactly k iterations, they become impractical as the problem dimensions (k,m,n) grow. The combination of ideas from MP and HT lead one to Iterative Hard Thresholding [7] whose per iteration complexity is higher, but which has the advantage of potentially converging in far fewer than k iterations. A further improvement to IHT which employs an optimal step in the proposed subspace leads to Normalized Iterative hard Thresholding [8]. Finally, combining the orthogonalization of OMP with the selection of a full candidate support set in HT leads to two-stage greedy algorithms such as Compressive Sampling Matching Pursuit (CoSaMP) [30], Subspace Pursuit (SP) [13], or Hard Thresholding Pursuit (HTP) [22]. These two-stage greedy algorithms incorporate a least squares problem in each iteration which further increases the per iteration complexity but potentially decreases the number of required iterations. These algorithms are more fully discussed in Section 2.

1.2 GPU Computing in Sparse Approximation

Even with the greedy algorithms outlined above, the total complexity of the algorithms require significant computation time when executed as a serial algorithm on a central processing unit (CPU). In each iteration of each of the aforementioned algorithms, at least one matrix-vector multiplication and one support set identification is required. When the problem dimensions are large, the number of iterations causes the standard CPU implementations to be too slow for practical application or large-scale testing. With the 2010 introduction of the Fermi architecture on Nvidia's GPU designed for high-performance computing, a GPU implementation will be trustwor-

thy, accurate, and provide a significant acceleration. In this paper, we describe our implementation of the Hard Thresholding family of greedy algorithms in the form of a CUDA based heterogeneous CPU-GPU software *GPU Accelerated Greedy Algorithms (GAGA)* [5]. This software is designed to provide a platform for large-scale testing of greedy algorithms on problem dimensions ranging from $n = 2^{10}$ to $n = 2^{20}$, although it is also capable of directly solving a specific problem. The GPU implementation reveals that the parallelized matrix-vector multiplications place a new emphasis on efficiently identifying the support set in each iteration. The GAGA implementation is described in Section 2 with an emphasis on the methods used to accelerate the support set identification. Section 4 provides an experimental performance analysis of the software where we observe up to 70x acceleration over standard MatLab central processing unit implementations using automatic multi-threading.

1.3 Notation

Throughout the paper, (k,m,n) define the compressed sensing problem dimensions of recovering a k-sparse vector of length n from m measurements. The measurements, y, come from the application of an $m \times n$ matrix A to the initial vector x, $y = Ax$. The algorithms discussed in the paper return an approximation $\hat{x}$ to the initial vector x. We will let T denote a set of indices which determine the active components of a vector or matrix. For example, the matrix whose columns are indexed by T is denoted A_T while x_T denotes the active components in the vector. The transpose of the matrix A is denoted A^* while the pseudo inverse is $A^\dagger$.

Specific names of functions from the software are designated with the typewriter font, such as `function`. Variables in the pseudo code and in the text are written in italics such as *vec* or vec_d. Here, the subscript d denotes that this variable is exclusively on the device, a common moniker for the GPU. A subscript of h, e.g. vec_h, denotes that the variable lives on the CPU or host (for example see Subrout. 3, Line 7). A CUDA kernel is called with triple angle brackets as in the actual code including the scheduling parameters, `kernel` $<<< blocks, threads >>>$ (arguments).

2 Algorithms

We have implemented six greedy algorithms for the solution of (1), namely two variants of Hard Thresholding: ThresholdSD and ThresholdCG; a fixed step size iterative hard thresholding, IHT; normalized iterative hard thresholding, NIHT; hard thresholding pursuit, HTP; and a two stage projection algorithm that is essentially equivalent to CoSaMP and Subspace Pursuit, CSMPSP. Each of these algorithms follows a similar basic skeleton: initialization and initial support detection and an iterative process of updating approximations until a stopping criteria is met. The algorithms differ in the method by which they update the approximations in each iteration.

The algorithms are listed in a pseudo code (Alg. 1-6) formulation using the following generic functions which are shared by several of the algorithms. The details of the implementations of these generic functions are described in the subsequent subsections.

- $T = \texttt{DetectSupport}(x)$ returns the index set, T, of the k largest magnitude entries[1] of the vector x.
- $x = \texttt{Threshold}(x,T)$ is a hard thresholding operation setting each entry of x to zero if the index of that entry is not an element of T.
- $x = \texttt{RestrictedSD}(x,T,y)$ performs a single step of steepest descent restricted to the subspace indexed by T, i.e. this performs a steepest descent step for the proxy problem $y = A_T x_T$.
- $x = \texttt{RestrictedCG}(x,T,y)$ performs a single step of the conjugate gradient method restricted to the subspace indexed by T, i.e. this performs a conjugate gradient step for the proxy problem $A_T^* y = A_T^* A_T x_T$.
- $x = \texttt{RSDProjection}(x,T,y)$ computes $x_T = A_T^{\dagger} y$ via a subspace restricted steepest descent, i.e. the values of x indexed by T are obtained via a steepest descent projection restricted to the subspace indexed by T while the remaining values of x are set to zero.
- $x = \texttt{RCGProjection}(x,T,y)$ computes $x_T = A_T^{\dagger} y$ via a subspace restricted conjugate gradient method, i.e. the values of x indexed by T are obtained via a conjugate gradient projection restricted to the subspace indexed by T while the remaining values of x are set to zero.

Algorithm 1 ThresholdSD (Hard Thresholding with Steepest Descent)

Input: A, y, k
Output: A k-sparse approximation $\hat{x}$ of the target signal x

Initialization and Initial Support Detection

1: $x_0 = A^* y$
2: $T_0 = \texttt{DetectSupport}(x_0)$
3: $x_0 = \texttt{Threshold}(x_0, T_0)$

Approximation:

1: $\hat{x} = \texttt{RSDProjection}(x_0, T_0, y)$
2: **return** $\hat{x}$

2.1 Subroutines

In this section, we describe the subroutines and CUDA kernels used as standard building blocks for the implementation of our greedy algorithms. These subroutines are inherent to most greedy algorithms and can be used to somewhat painlessly expand the software to include other greedy algorithms. The main subroutines are listed at the beginning of Sec. 2. They are divided into three topics: hard thresholding (`Threshold` and `DetectSupport`), subspace restricted numerical linear algebra (`RestrictedSD` and `RestrictedCG` with subspace projections based on these functions), and matrix-vector multiplications (subsampled discrete cosine transform, sparse matrix-vector multiplication, and a generic matrix-vector multiplication).

[1] In our implementation, the number of elements in the index set is not necessarily exactly k based on the method of support set detection. This is discussed in detail in Sec. 2.1.

Algorithm 2 ThresholdCG (Hard Thresholding with Conjugate Gradient)

Input: A, y, k
Output: A k-sparse approximation $\hat{x}$ of the target signal x

Initialization and Initial Support Detection

1: $x_0 = A^*y$
2: $T_0 = \texttt{DetectSupport}(x_0)$
3: $x_0 = \texttt{Threshold}(x_0, T_0)$

Approximation:

1: $\hat{x} = \texttt{RCGProjection}(x_0, T_0, y)$
2: **return** $\hat{x}$

Algorithm 3 IHT (Iterative Hard Thresholding [7])

Input: A, y, k, ω (a fixed step size)
Output: A k-sparse approximation $\hat{x}$ of the target signal x

Initialization and Initial Support Detection

1: $x_0 = A^*y$
2: $T_0 = \texttt{DetectSupport}(x_0)$
3: $x_0 = \texttt{Threshold}(x_0, T_0)$
4: $r_0 = y - Ax_0$

Iteration: During iteration l, **do**

1: $x_l = x_{l-1} + \omega A^* r_{l-1}$
2: $T_l = \texttt{DetectSupport}(x_l)$
3: $x_l = \texttt{Threshold}(x_l, T_l)$
4: $r_l = y - Ax_l$
5: Update Stopping Parameters
6: **if** Stopping Criteria **then**
7:     **return** $\hat{x} = x_l$
8: **else**
9:     Perform iteration $l+1$
10: **end if**

Algorithm 4 NIHT (Normalized Iterative Hard Thresholding [8])

Input: A, y, k
Output: A k-sparse approximation $\hat{x}$ of the target signal x

Initialization and Initial Support Detection

1: $x_0 = A^*y$
2: $T_0 = \texttt{DetectSupport}(x_0)$
3: $x_0 = \texttt{Threshold}(x_0, T_0)$

Iteration: During iteration l, **do**

1: $x_l = \texttt{RestrictedSD}(x_{l-1}, T_{l-1}, y)$
2: $T_l = \texttt{DetectSupport}(x_l)$
3: $x_l = \texttt{Threshold}(x_l, T_l)$
4: Update Stopping Parameters
5: **if** Stopping Criteria **then**
6:     **return** $\hat{x} = x_l$
7: **else**
8:     Perform iteration $l+1$
9: **end if**

Algorithm 5 HTP (Hard Thresholding Pursuit [22])

Input: A, y, k
Output: A k-sparse approximation $\hat{x}$ of the target signal x

Initialization and Initial Support Detection

1: $x_0 = A^* y$
2: $T_0 = \texttt{DetectSupport}(x_0)$
3: $x_0 = \texttt{Threshold}(x_0, T_0)$

Iteration: During iteration l, **do**

1: $x_l = \texttt{RestrictedSD}(x_{l-1}, T_{l-1}, y)$
2: $T_l = \texttt{DetectSupport}(x_l)$
3: $x_l = \texttt{Threshold}(x_l, T_l)$
4: $x_l = \texttt{RCGProjection}(x_l, T_l, y)$
5: Update Stopping Parameters
6: **if** Stopping Criteria **then**
7: **return** $\hat{x} = x_l$
8: **else**
9: Perform iteration $l+1$
10: **end if**

Algorithm 6 CSMPSP (CoSaMP [30], Subspace Pursuit [13])

Input: A, y, k
Output: A k-sparse approximation $\hat{x}$ of the target signal x

Initialization and Initial Support Detection

1: $x_0 = A^* y$
2: $T_0 = \texttt{DetectSupport}(x_0)$
3: $x_0 = \texttt{Threshold}(x_0, T_0)$
4: $x_0 = \texttt{RCGProjection}(x_0, T_0, y)$
5: $r_0 = y - Ax_0$

Iteration: During iteration l, **do**

1: $S_l = \texttt{DetectSupport}(A^* r_{l-1})$
2: $\Lambda_l = T_{l-1} \cup S_l$
3: $x_l = \texttt{RCGProjection}(x_{l-1}, \Lambda_l, y)$
4: $T_l = \texttt{DetectSupport}(x_l)$
5: $x_l = \texttt{Threshold}(x_l, T_l)$
6: $r_l = y - Ax_l$
7: Update Stopping Parameters
8: **if** Stopping Criteria **then**
9: **return** $\hat{x} = x_l$
10: **else**
11: Perform iteration $l+1$
12: **end if**

2.1.1 Support Detection and Hard Thresholding

The algorithms included in this software all require a hard thresholding of magnitude k, where the entries of a vector with the k largest magnitudes are preserved while all other entries are set to zero. An obvious method for performing this task is to identify the kth largest magnitude in the vector and then simply set every entry to zero if its magnitude is smaller than the kth largest magnitude. However, the algorithms require

more than just the threshold; they also require the indices identifying the set of the k largest magnitudes in the vector. We call this index set the *support set* of the vector. If one knows the kth largest magnitude, identifying the support set is straightforward by simply recording which elements have a larger magnitude in an additional support vector.

Standard implementations of these recovery algorithms employ a sorting routine in order to extract both the kth largest magnitude and the support set. In a traditional serial (CPU) implementation, the cost of the matrix-vector multiplications are significantly greater than that of identifying the support set. As such, the apparent waste in sorting the vector is mitigated by its relative cost to the matrix-vector multiplication. In our parallel implementations, the computational cost of the matrix-vector multiplications are dramatically reduced, and the computational costs for support detection via sorting and a matrix-vector multiplication are now on par. Thus, it is important to reduce the cost of support detection in our GPU implementation.

Subroutine 1 `FindSupport_sort`

Input: vec_d, $support_d$, k
Output: A thresholded version of vec_d and the support identification vector $support_d$.

1: `thrust::sort`$($`abs`$(vec_d))$
2: $b =$ `abs`$(vec_d[n-k])$
3: `one_vector` $<<< blocks, threads >>> (supp_d)$
4: `threshold_and_support` $<<< blocks, threads >>> (vec_d, support_d, b)$

`one_vector`: (CUDA Kernel)
Input: $support_d$
1: Set id as the thread identifier
2: $support_d[id] = 1$

`threshold_and_support`: (CUDA Kernel)
Input: vec_d, $support_d$, b
1: Set id as the thread identifier
2: **if** $($`abs`$(vec_d[id]) < b)$ **then**
3: $vec_d[id] = 0$
4: $support_d[id] = 2$
5: **end if**

For comparative and expository purposes, we have implemented the sorting method of support detection and thresholding and describe it here. Sorting algorithms for GPU computing are now rather mature with the most efficient sorting routine a radix sort by Merrill and Grimshaw [27]. This highly efficient radix sort is now included as `thrust::sort` in the Thrust library [23] which is included in the CUDA 4.0 release [31]. Once the absolute value vector is sorted, the kth largest magnitude is easily extracted. The act of thresholding and identifying the support set can be accomplished simultaneously. For consistency with our default implementation, the support identification vector used with sorting is an integer vector with a 1 identifying a member of the support and a 2 identifying an off-support element.

One common method for improving performance over sorting is to use a k-selection algorithm which reduces the computational complexity of identifying the

kth largest element. At the beginning of our work on this software, no selection algorithm implementations existed for the GPU which were superior in performance to `thrust::sort`. In 2011, four GPU k-selection algorithms were introduced [1,3,28] which could be used to replace the sorting step in Subrout. 1. While these selection algorithms gain a significant advantage over `thrust::sort` as the vector gets very large they have essentially equivalent performance for vectors of length 2^{20} or less.

Our method to identify the support set is an approximate k-selection based on a linear projection of the entries in the vector into linearly spaced bins, similar to a selection based on distributive partitioning [2]. The approximate k-selection technique requires only a single examination of the entries in the vector and is amenable to parallelization in that the linear projection of each entry in the vector requires no communication with any other element. Furthermore, this method increases the likelihood of identifying the correct support in a given iteration as it permits elements of the vector with magnitude slightly smaller than the kth largest magnitude to be included in the support.

The support set is identified by projecting the elements of the vector, vec_d, into bins. The bins are determined by linearly partitioning the interval $[0, \mathtt{max}(\mathtt{abs}(vec_d))]$ into a certain number of bins, $numBins = \mathtt{max}(1000, n/20)$ where n is the length of the vector. As the elements of the vector are projected into the bins a counter is updated in a separate vector $bincounter_d$. This is implemented as the CUDA kernel `LinearBinning` described in Subrout. 3 where we note that larger magnitude entries are assigned to smaller numbered bins. (In other words, the largest element is assigned to bin 0 and the smallest to bin $numBins-1$.) At this point, a cumulative sum of the counter vector is performed to identify which bin contains the kth largest magnitude; we call this bin $kbin$. Now the set of entries in the vector whose magnitude is larger than the kth largest has been identified along with any entry whose magnitude is close enough in value to the kth largest magnitude to fall in $kbin$.

Subroutine 2 `Threshold`(CUDA Kernel)

Input: vec_d, bin_d, $kbin$
Output: A thresholded version of vec_d.

1: Set id as the thread identifier
2: **if** $(bin_d[id] > kbin)$ **then**
3: $\quad vec_d[id] = 0$
4: **end if**

The support set is now identified by the bin vector bin_d and the identifier $kbin$. The support set can be used in any subsequent function which requires restriction to the support. For example, we frequently want to hard threshold a vector to the support set. This is described in Subrout. 2.

The serial nature of counting suggests that we count the entries in each bin using atomic functions, in particular `atomicAdd`. When two or more threads wish to increment the count at precisely the same time, atomic functions eliminate the conflict by forming a queue for the counter. If a single bin contains a large number of elements, the long queue causes a degradation in performance. Our first decision is to refuse

Subroutine 3 `FindSupport`

Input: vec_d, $bins_d$, $bincounter_d$, $bincounter_h$, k, α, $numBins$, $maxBin$, $minValue$, $maxChange$
Output: The support of roughly the k largest magnitude entries of vec_d in the form of a vector $bins_d$ and and a support identifier $kbin$.

```
 1: if (minValue > maxChange) then
 2:     minValue = minValue − maxChange
 3: else
 4:     max = maxMagnitude(vec_d)
 5:     slope = (numBins − 1)/max
 6:     LinearBinning <<< blocks, threads >>> (vec_d, bins_d, bincounter_d, k, maxBin, slope, max)
 7:     copyToCPU(bincounter_d, bincounter_h)
 8:     sum = 0, kbin = 0
 9:     while (sum < k) && (kbin < maxBin) do
10:         sum = sum + bincounter_h[kbin]
11:         kbin = kbin + 1
12:     end while
13:     kbin = kbin − 1
14:     if (sum < k) then
15:         α = .5α
16:         maxBin = ⌊(1 − α)numBins⌋
17:     end if
18:     minValue = max − (kbin + 1)/slope
19: end if
```

`LinearBinning`: (CUDA Kernel)
Input: vec_d, $bins_d$, $bincounter_d$, k, $maxBin$, $slope$, max

```
 1: Set id as the thread identifier
 2: temp = abs(vec_d[id])
 3: if (temp > 10^−6 max) then
 4:     bin_d[id] = ⌊slope ∗ (max − temp)⌋
 5:     if (bin[id] < maxBin) then
 6:         atomicAdd(bincounter_d[bin_d[id]], 1)
 7:     end if
 8: else
 9:     bin_d[id] = ⌈slop ∗ max⌉ + 1
10: end if
```

to count entries whose magnitude is less than 10^{-6} times the maximum magnitude in the vector. In fact, we assign these to a dummy bin which is never counted. Second, in early iterations the error in our approximation in the vector produces a large number of small values in the off-support entries after a matrix-vector multiplication. However, as the algorithms progress toward convergence, the errors from the off-support matrix-vector multiplication begin to vanish and the correct support begins to separate from the small values caused by these errors. Also, since the algorithms are iterative, it is perfectly acceptable to first identify a subset of the support and then proceed to expand the proposed support to the appropriate size in subsequent iterations. Our implementation does just this by introducing a parameter $\alpha \in [0,1)$ which reduces the number of bins to be counted. Rather than count every bin we count only a subset of the bins which will contain the largest magnitudes. Since our linear projection labels the bins containing the largest magnitudes with the smaller indices, we do so by setting a maximum bin and count only those bins up to the maximum

bin; let $maxBin = (1-\alpha)numBins$ so that only the first $(1-\alpha)$ fraction of the bins are counted. The default setting for α is 0.25. If the size of the proposed support is smaller than k, α is reduced by a factor of 0.5 and if needed will ultimately converge to counting every bin. This ensures that the final support set will have at least k entries. If one wishes to enforce identification of at least k elements in the support at every iteration one can simply set $\alpha = 0$.

Finally, we also impose a means of skipping the support identification whenever possible which we refer to as *dynamic support detection*. In each iteration the algorithms provide information about how a vector was updated. Using this information allows us to answer the question, “Is it possible that the support set has changed?” If not, we simply skip the support identification altogether. Answering this question exactly in each iteration introduces an unnecessary computational cost as it would require finding the minimum magnitude in the support set. However, a proxy to the minimum value in the support, namely the endpoint of the bin containing the kth largest magnitude (stored as *minValue*), is provided for free. Rather than computing the change in each entry, we simply identify the largest possible change in the update step, *maxChange*. The worst case situation is that the smallest magnitude on the support was decreased by *maxChange* while an entry off of the support was increased by *maxChange*. Hence, if $minValue > 2maxChange$, the support could not have changed and we skip the support identification. For the subsequent iteration, we simply enforce the worst case update by reducing *minValue* by $2maxChange$. In the early iterations when the algorithm is searching for the correct vector, the support detection step is implemented frequently. However, a considerable savings is realized in later iterations when the algorithm is converging to a solution and the updates rarely change the support.

A full description of our dynamic support detection, `FindSupport`, is detailed in Subrout. 3 where *maxChange* already includes the scaling by 2 from elsewhere, for example Subrout. 4, Step 6. Clearly, dynamic support detection can also be implemented when using sorting or selection to find the kth largest magnitude. We omit the pseudo code in this paper but have implemented and tested these ideas in the software. Performance comparisons for sorting, sorting with dynamic support detection, and `FindSupport`, both with $\alpha = 0.25$ and $\alpha = 0$, are available in Sec. 4.

2.1.2 Subspace Restricted Iterative Numerical Linear Algebra

After the work to identify the support set has been completed in a given iteration, we operate under the belief that this is the correct support set. In other words, we believe that any update to the vector should occur in the lower dimensional subspace associated with the support set rather than in the full ambient n-dimensional space. Again, we represent the support set, called T in the early part of Sec. 2, with the vector of bin assignments, $bins_d$ and $kbin$, the identity of the bin containing the kth largest element. The algorithms in this software rely on iterative numerical linear algebra routines, the method of steepest descent and the conjugate gradient method. For example, IHT chooses the search direction as the negative gradient, computed as $grad = A^*(resid) = A^*(y - Ax)$. This is of course the correct direction for an optimal step, but IHT performs this step with a fixed step size. While this works often, it is

not the optimal step size and can cause the algorithm to converge very slowly or even to fail completely.

Subroutine 4 `RestrictedSD`

Input: vec_d, $bins_d$, $kbin$
Output: The updated vector vec_d and the worst case change to vec_d from this update.

1: $resid = y - A(vec_d)$
2: $grad = A^*(resid)$ (steepest descent direction)
3: $grad_thresh = \texttt{Threshold}(grad, bins_d, kbin)$ (restrict the SD direction to the subspace)
4: $\mu = \frac{\|grad_thresh\|_2^2}{\|A(grad_thresh)\|_2^2}$ (optimal step size in restricted subspace)
5: $vec_d = vec_d + \mu grad$ (update via the restricted SD step)
6: $maxChange = 2\mu\texttt{max}(\texttt{abs}(grad))$ (record *maxChange* for dynamic support detection)

Subroutine 5 `RSDProjection`

Input: vec_d, $bins_d$, $kbin$, y
Output: The projection of y onto the subspace defined by $bins_d$ and $kbin$.

Initialization:

1: $vec_d = A^*y$ (use the transpose as an approximate inverse)
2: $(bins_d, kbin) = \texttt{FindSupport}(vec_d, bins_d, k, \text{control parameters})$ (find the support set)
3: $vec_d = \texttt{Threshold}(vec_d, bins_d, kbin)$ (restrict the vector to the support set)

Projection:

1: **while** stopping criteria are not met **do**
2: $vec_d = \texttt{RestrictedSD}(vec_d, bins_d, kbin)$ (one restricted steepest descent step)
3: $vec_d = \texttt{Threshold}(vec_d, bins_d, kbin)$ (threshold to the support set)
4: update the stopping parameters
5: **end while**
6: **return** vec_d

NIHT also chooses the gradient direction based on the residual from our approximation which has been thresholded to our support set. Rather than take a fixed-size step, NIHT takes the optimal single step under the assumption that we have found the correct support set. In other words the step size is computed only in our restricted subspace defined by our support set, $T := (bins_d, kbin)$. In each iteration of NIHT, a single step of steepest descent is performed to update the vector. This subspace restricted steepest descent is implemented as the function `RestrictedSD`. In Subrout. 4 we outline the mathematics of a restricted steepest descent step. The actual CUDA based implementation relies heavily on `cuBLAS`, the GPU-accelerated version of the complete standard Basic Linear Algebra Subroutines (BLAS) library.

An alternative to the update in NIHT is to compute a projection of y on the chosen subspace via a pseudo inverse, $x_T = A_T^\dagger y$. For example, in `ThreshholdSD`, the support is detected only one time and then the measurement vector y is projected onto this set. One method for iteratively accomplishing this projection is to iteratively run `RestrictedSD` on the chosen support set until some stopping criterion is

met. In our implementation, we did not explicitly write the function `RSDProjection` (Subrout. 5), but we include it here for reference from Alg. 1.

Subroutine 6 `RestrictedCG`

Input: vec_d, $grad$, $grad_previous$, $bins_d$, $kbin$
Output: The updated vector vec_d and the historical direction $grad_previous$.

1: $resid = A(grad)$
2: $\mu = \frac{\|grad_previous\|_2^2}{\|resid\|_2^2}$ (optimal step size in restricted subspace)
3: $vec_d = vec_d + \mu grad$ (update via the restricted CG step)
4: $grad_new = A^*(resid)$ (determine new CG direction)
5: $grad_new = \texttt{Threshold}(grad_new, bins_d, kbin)$ (restrict CG direction to the subspace)
6: $grad_new = grad_previous + \mu grad_new$ (update new CG direction in the restricted subspace)
7: $\beta = \frac{\|grad_previous\|_2^2}{\|grad_new\|_2^2}$
8: $grad = grad_new + \beta grad$ (update next CG direction in the restricted subspace)
9: $grad_previous = grad_new$ (record the previous CG direction)

Subroutine 7 `RCGProjection`

Input: vec_d, $bins_d$, $kbin$, y
Output: The projection of y onto the subspace defined by $bins_d$ and $kbin$.

Initialization:
1: $vec_d = A^* y$ (use the transpose as an approximate inverse)
2: $(bins_d, kbin) = \texttt{FindSupport}(vec_d, bins_d, k, \text{control parameters})$ (find the support set)
3: $vec_d = \texttt{Threshold}(vec_d, bins_d, kbin)$ (restrict the vector to the support set)

Initial CG Direction:
1: $grad = A^*(y - A(vec_d))$ (determine initial gradient direction)
2: $grad = \texttt{Threshold}(grad, bins_d, kbin)$ (restrict the gradient to the support set)
3: $grad_previous = grad$

Projection:
1: **while** stopping criteria are not met **do**
2: $(vec_d, grad, grad_previous) = \texttt{RestrictedCG}(vec_d, grad, grad_previous, bins_d, kbin)$
3: update the stopping parameters
4: **end while**
5: **return** vec_d

The two-stage greedy algorithms, such as HTP or CSMPSP (CoSaMP/Subspace Pursuit), also require a projection in each iteration. As is well known, convergence can be accelerated by using the conjugate gradient method (CG). Here we discuss the projection on a support set using a restricted CG. For consistency in our implementation, we include a function, `RestrictedCG` (Subrout. 6), which performs a single conjugate gradient (CG) step in the restricted subspace. To perform the projection, we wrap the single step of CG in a while loop and detail this procedure as `RCGProjection` in Subrout. 7.

Unlike steepest descent, a CG step requires information pertaining to the previous step. For simplicity in the code, the initial conjugate gradient direction is computed

mat	*A_mat*	*AT_mat*
gen	(out, in, A, m, n)	(out, in, A, m, n)
dct	$(out, in, rows, m, n)$	$(out, in, rows, m, n)$
smv	$(out, in, rows, cols, vals, m, n, nz)$	$(out, in, rows, cols, vals, m, n, nz)$

Table 1 Input arguments for matrix-vector multiplication and transpose matrix-vector multiplications. (For *smv*, *nz* denotes the total number of nonzero entries.)

outside of this function as $grad = A^*(y - A(vec_d))$ while the function returns the most recent direction as *grad_previous*. Hence, the history is passed to the function via *grad* and *grad_previous*. All of the vectors, vec_d, *grad*, *grad_previous*, must already have been thresholded to the restricted subspace when passed to `RestrictedCG`. As this single CG step is used only in projection in our current implementation, it can not determine the maximum change to the vector in the update step. In fact, if CG is permitted to converge in a restricted subspace, this CG projection will provide the minimum ℓ_2 solution in this subspace. Hence, if the support set has been correctly identified, the algorithm returns the correct solution.

2.1.3 Matrix-vector Multiplication

The greedy algorithms, in particular the subspace restricted numerical linear algebra, rely heavily on matrix-vector products. This software deals with the matrix-vector multiplication in three ways. First of all, any complete matrix can be passed to the algorithms which we refer to as a *generic* matrix-vector multiplication, or *gen*. To model a fast matrix-vector multiplication, we use the discrete cosine transform, *dct*. Finally, a sparse matrix enjoys some advantages in multiplication and thus we have implemented a *sparse mat-vec*, *smv*.

The matrix and transpose multiplications are written so that one may easily call the multiplication in the form of a function: $A_mat(out, in, \text{matrix info})$. The function *A_mat* performs a matrix-vector multiplication for the matrix of type *mat* where *mat* is one of $\{gen, dct, smv\}$. If A is a matrix of type *mat*, then providing the appropriate information about the matrix to *A_mat* will complete the matrix-vector multiplication $out = A(in)$. In compressed sensing, we are dealing with matrices with fewer rows than columns. Thus, the transpose multiplication is equally important. Optimizing a certain type of multiplication along the rows would not pay off in the long run as there are roughly an equal number of matrix transpose multiplications as matrix multiplications. The transpose multiplications $out = A^*(in)$ are executed with functions $AT_mat(out, in, \text{matrix info})$.

Clearly, each of these matrix-vector multiplications have their own input requirements. The generic matrix-vector multiplication simply takes the complete matrix A and its dimensions m, n as the input for the matrix information. The subsampled DCT requires the dimensions m, n and the subset of rows from the full $n \times n$ DCT which form our $m \times n$ matrix. Finally the sparse mat-vec is given the dimensions m, n, the locations and values of the nonzero elements in the form of the rows, columns, and values, and finally the total number of nonzeros. Table 1 details the input and output parameters for these functions.

2.2 Problem Generation

Although the functions in this software package are capable of taking a problem directly from MatLab, the primary motivation for building this software is the ability to perform large-scale testing on randomly generated problems. To create a random problem for compressed sensing one must create both a random input vector, *vec_input*, and a random matrix, A. A random k-sparse vector requires both a random choice for the values of the entries and random support set for these nonzero values. In standard MatLab implementations this involves not only the generation of a large quantity of random numbers, but support sets are generally chosen with a sorting based selection routine. The random problem generation package in this software exploits both the parallel generation of pseudo random numbers via the built-in function `cuRAND` and the computational savings from our GPU support detection, `FindSupport`. Similar computational savings are obtained for the generation of the matrix A.

2.2.1 Randomly Generated Input Vectors

The input vectors for testing the greedy algorithms are obviously k-sparse vectors of length n. One must randomly generate both the locations and values for the k nonzero entries in the vector. Due to `cuRAND`'s rapid ability to generate pseudo random numbers, this is accomplished by generating $2n$ random numbers, and then using the first half to determine the locations and the other half to determine the values. To identify the support set, n uniformly distributed random numbers are generated and the random support set is identified by finding the locations of the k largest values in this vector of length n via our support detection function `FindSupport`. The remaining n random numbers are determined according to an input parameter which chooses the distribution from which the values are drawn. The software is currently equipped to create a vector with entries from three distributions: a sign pattern $\{-1,1\}$, a uniform distribution $\mathscr{U}(0,1)$, and a Gaussian distribution $\mathscr{N}(0,1)$. Using the support set obtained from the other n random numbers, the vector of values is thresholded to this support set and recorded as the input vector. This is outlined in Subrout. 8.

Subroutine 8 Random Vector Creation

Input: k, n, *vecDistribution*

Output: The k-sparse vector *vec_input* with random support and random values from *vecDistribution*.

1: $vec_support = \texttt{cuRAND}(n, \mathscr{U}(0,1))$ (n random numbers for support creation)
2: $(bins_d, kbin) = \texttt{FindSupport}(vec_support, bins_d, k,$ control parameters$)$ (find the support set)
3: $vec_input = \texttt{cuRAND}(n, vecDistribution)$ (n random values from *vecDistribution*)
4: $vec_input = \texttt{Threshold}(vec_input, bins_d, kbin)$ (create k-sparse vector)
5: **return** *vec_input*

2.2.2 *Randomly Generated Measurement Matrices*

As discussed in Sec. 2.1.3, the software is equipped with three variants of matrix-vector multiplication, the randomly subsampled discrete cosine transform (DCT), sparse matrices (SMV), and a generic matrix (gen).

Generating the Rows for the DCT. Randomly selecting the rows for the DCT requires the generation of n random numbers which is performed on the GPU using `cuRAND`. A vector of row indices, integers from 1 to n, is also created on the GPU. We then randomly shuffle the index set on the host by transferring both the n random values from $\mathscr{U}(0,1)$, *rand_values*, and the index set back to the host. A straightforward shuffle is executed by swapping the index in position j with the index in position $\lfloor j * rand_values[j] \rfloor$. The m indices are then transferred back to the device.

Executing a shuffle of m integers on the host requires exactly m multiplications, $3m$ reads, and $2m$ writes plus the transfer of the data from the device to the host and back again. At first glance, this might seem inefficient and one might instead wish simply execute `FindSupport` on the n random values to determine the locations of the largest m values. This would certainly return a random selection of rows. Since the number of rows ranges from 1 to n, $1 \leq m \leq n$, the use of `FindSupport`, which is also a heterogeneous function, is in fact burdensome as $m \to n$. For small values of m the performance of these two routines is comparable while the CPU shuffle outperforms the use of `FindSupport` as $m \to n$. Executing a shuffle on the device would require the use of an atomic function to avoid duplication of a single row in the selection. The GPU shuffle is advantageous for large m but the CPU shuffle is superior for small m. Our use of the CPU shuffle is a compromise between these competing strategies over the full range of m.

Generating a Sparse Matrix. The sparse matrices available for random generation have a fixed number, p, of nonzero entries per column[2]. There are two sparse matrix ensembles, namely every nonzero has the value 1 or the entries have a random sign pattern $\{-1,1\}$ followed by normalizing the columns by dividing by $\sqrt{p}$. Therefore we generate the values on the GPU in the vector *vals* of length np where n is again the number of columns of the matrix. The only remaining task is to randomly assign the p rows in each of the n columns which hold the nonzero values. This is easily performed by creating a vector *rows* of length np. For each segment of length p in *rows* we assign the row index $\lfloor m * rand_values[j] \rfloor$ where j is the index of the vector *rows* and *rand_values* is a vector populated with np entries from $\mathscr{U}(0,1)$. The implementation actually generates more than np random values and checks to ensure that a single column is actually assigned p distinct rows indices for the nonzeros. If the random assignment of rows fails to generate p distinct rows for each column an error flag is returned and the test is aborted.

[2] The software does not require a fixed number of nonzeros per column when passing a known problem using a sparse matrix directly to the algorithms.

Generating a Generic Matrix. The software creates two variants of a generic matrix, a Gaussian matrix with entries populated from $\mathcal{N}(0,m^{-1})$ and a dense matrix with entries drawn from a normalized sign pattern $\{-1/\sqrt{m}, 1/\sqrt{m}\}$. To create an $m \times n$ matrix simply requires the creation of mn random values. For the Gaussian matrix the entries of the matrix are derived from `cuRAND`'s normal distribution. The alternative ensemble is created by forming mn elements with `cuRAND`'s uniform distribution, shifting by 0.5, and recording the sign of each entry.

2.2.3 Initial Measurement Vector

Finally, we must create the vector of measurements we wish to pass to the algorithms. Having created *vec_input* as described in Sec. 2.2.1 and an appropriate measurement matrix A_{mat} as described in Sec. 2.2.2, we simply perform the matrix-vector multiplication $y = A_{mat}(vec_input)$ as described in Sec. 2.1.3. One important aspect of testing compressed sensing algorithms is the consideration of noisy measurements. A set of problem generation functions which add noise to the measurements is included. A noise vector is produced by passing these functions a noise level parameter *noise_level*, creating m random values from $\mathcal{N}(0,1)$, scaling the noise vector to have ℓ_2 norm $noise_level \cdot \|y\|_2$, and adding this noise vector to the measurements y.

2.3 GPU Implementation

In CUDA, functions are written for the GPU in the form of *kernels* which take control parameters to seamlessly assign the parallel tasks to the cores. The work for a kernel is divided into blocks where each block is assigned the same number of threads. The maximum number of threads, *maxThreadsPerBlock*, for a given GPU is easily obtained with a built-in device query function from CUDA. The preponderance of kernels in this software execute a task on a single element of a vector. Thus, a vector of length L requires L threads. Empirically we observe that minimizing the number of blocks produces the best performance.

The vectors consider by the algorithms in this software have length L where L is one of m, n, mn, np, or *numBins*. Different scheduling parameters are determined based on the length of the vector passed to the kernel. The kernels are executed with the scheduling parameters *numBlocks* and *threadsPerBlock* where

$$threadsPerBlock = \min(L, maxThreadsPerBlock);$$
$$numBlocks = \left\lceil \frac{L}{threadsPerBlock} \right\rceil.$$

In this way, we are guaranteed at least L threads, and all kernels ensure that any thread with index greater than L remains idle.

Alternatively, one may adopt the convention of writing kernels so that the number of blocks is a multiple of the number of multiprocessors on the GPU. This requires writing kernels that execute tasks on multiple elements in a vector. On the surface, this ensures full parallelization by employing every core. Our convention has the

potential for assigning blocks to a fraction of the multiprocessors. However, as our kernels are executing a single task on a single element, we still achieve the same level of parallelization. Suppose the length L is not a multiple of the number of cores. Then the convention of assigning blocks as a multiple of the number of multiprocessors requires R threads which act on one more element than the other threads where

$$R = L - \left\lfloor \frac{L}{\text{number of multiprocessors}} \right\rfloor .$$

These R threads will perform the extra task on exactly R cores. Our convention of maximizing the number of threads will require the execution of $\tilde{R}$ threads on $\tilde{R}$ cores where

$$\tilde{R} = L - \left\lfloor \frac{numBlocks}{\text{number of multiprocessors}} \right\rfloor * (\text{number of multiprocessors}) * threadsPerBlock.$$

Even though R and $\tilde{R}$ may not be identical, they are both smaller than the number of cores and therefore require the full use of the GPU.

3 Software

The software, *GPU Accelerated Greedy Algorithms for Compressed Sensing* [5], is currently enabled to solve compressed sensing problems with five greedy algorithms: Thresholding, Iterative Hard Thresholding [7], Normalized Iterative Hard Thresholding [8], Hard Thresholding Pursuit [22], and CoSaMP/Subspace Pursuit [30, 13]. The software, written in CUDA-C, compiles Matlab executable files which define three functions to be executed as standard Matlab functions[3]. Since the source is written in CUDA-C, a user can readily alter the parent functions to create C/C++ executables rather than Matlab executables.

There are two main functionalities. First, the functions are capable of taking a problem directly from Matlab and employing the GPU to obtain the solution. This is especially useful for applications. The primary motivation for writing the software is the need for large-scale testing of these algorithms on large problems. Therefore, the software includes a significant testing suite for randomly generated problems. Information about the performance of the algorithm is written to a text file and Matlab scripts for reading and analyzing the output are also included. The random seed used to generate each random problem is included in the performance data so that each of the problems tested can be reproduced. For comparison to CPU performance, each function has an equivalent Matlab version which does not employ the GPU.

As described in Sec. 2.1.3, the software currently contains three matrix-vector multiplications, generic matrices (*gen*), subsampled discrete cosine transform (*dct*), and sparse matrices (*smv*). As these matrices require different input to describe the matrix, there are three distinct version of each of the main functions. In the following, we let $mat \in \{gen, dct, smv\}$ denote the suffix which describes the matrix-vector multiplication.

[3] While this software requires a CUDA enabled GPU, it is independent from Matlab and does not require the parallel processing toolbox.

The main functions compiled are gaga_dct, gaga_smv, and gaga_gen. A user may call these functions directly, or they can be accessed through a parent function gaga_cs. These functions are overloaded to have two main functionalities: testing and application. The algorithm testing variant generates a random problem to test a specified algorithm and matrix class and returns algorithm performance characteristics. The application variant returns an estimated sparse solution to a specific problem passed from the Matlab workspace with the sparse solution obtained by applying a user specified algorithm along with the measurement matrix, measurements, and algorithm options.

When used to test algorithm performance on randomly generated problems[4], the function automatically generates the problem using options *matrixEnsemble* and *vecDistribution* to specify the distribution of the random $m \times n$ measurement matrix and random k nonzeros in the sparse vector measured. The algorithm then solves the generated problem and returns to the Matlab workspace the following algorithm performance characteristics: errors, times, iterations, support identification, convergence rate, and the output of the algorithm. The function also records more detailed information to a date stamped text file. The input and output arguments for executing these functions are given in Tab. 2.

	Random Problem	
matrix class	output	input
gen	$[errors, times, iterations, checkSupport, convRate, \hat{x}]$	gaga_cs('alg', 'gen', $k, m, n, options$)
smv	$[errors, times, iterations, checkSupport, convRate, \hat{x}]$	gaga_cs('alg', 'smv', $k, m, n, p, options$)
dct	$[errors, times, iterations, checkSupport, convRate, \hat{x}]$	gaga_cs('alg', 'dct', $k, m, n, options$)

Table 2 Input arguments and output for generating and solving a random problem with gaga_cs. (For *smv*, p denotes the number of nonzero entries per column.) The *options* argument may be omitted in which case all variables take on their default values.

The fuction gaga_cs is also capable of taking a problem directly from Matlab, passing it to the GPU, and returning the solution to the Matlab workspace. In this case, there is no data written to a text file and the output to Matlab is the approximate solution vector, iterations, and convergence rate. The function always requires inputs specifying the algorithm to be used to recover the sparse vector, the type of the matrix that is being passed, the number of nonzeros k in the output vector, and the vector of measurements y. When passing a general matrix, class *gen*, the matrix A is passed directly. Sparse matrices, class *smv*, are passed in coordinate list (COO) format of *rows*, *cols*, and *values*. For discrete cosine transform matrices, class *dct*, the measurement matrix is passed by specifying its size $m \times n$ and the subset *rows* of the full discrete cosine transform matrix.

Admissible values for *alg* are the following strings: 'ThresholdSD', 'ThresholdCG', 'IHT', 'NIHT', 'HTP', and 'CSMPSP'. Algorithms 'NIHT' and 'HTP' support a further 'timing' option (described below). Options are set using the gagaOptions function

[4] The random sparse matrix generator is designed to be fast for $p \ll m$, and outside this regime can fail to construct the sparse matrix appropriately, resulting in a warning and the function terminating.

	Direct Problem	
matrix class	output	input
gen	$[\hat{x}, iterations, convRate]$	gaga_cs('alg', 'gen', $k, y, A, options$)
smv	$[\hat{x}, iterations, convRate]$	gaga_cs('alg', 'smv', $k, y, rows, cols, values, options$)
dct	$[\hat{x}, iterations, convRate]$	gaga_cs('alg', 'dct', $k, m, n, y, rows, options$)

Table 3 Input arguments and output for solving a direct problem with gaga_cs.

by passing pairs indicating a variable name followed by its specified value (which may be an integer, float, or string). For example,

```
>> options = gagaOptions('tol',0.0001,'maxiter',400,'vecDistribution','gaussian');
>> [err,tim,itr,supp,cnv,xout] = gaga_cs('NIHT','dct', k, m, n, options);
```

will generate a random problem with a $m \times n$ subsampled DCT matrix and a vector with k nonzeros from $\mathcal{N}(0,1)$, and then solve the problem using NIHT, with stopping criteria depending on a tolerance of 0.0001 and a maximum of 400 iterations. Options should be set using gagaOptions for typecasting and necessary ordering. A complete list of admissible options is presented in Tab. 4.

To generate timings data for performance comparisons of the subroutines, when the timing option is active, the function gaga_cs produces additional output to a text file reporting timings per iteration of FindSupport, RestrictedSD, and RCGProjection. Since the subroutines are consistent across the algorithms, timing the subroutines is available for NIHT and HTP.

The software package includes a mirror copy of the Matlab GPU software implementation written exclusively in Matlab using only the CPU. The primary purpose of this duplication of the code is to compare the performance of the GPU enabled functions to CPU only equivalent, and as a secondary feature to serve as a template to the CUDA-C code for those unfamiliar with this language. This software is included in the directory GAGA/gaga_matlab with the main function gaga_matlab_cs having the same input and output as given in Tab. 2 for random problem generation and evaluation. The CPU only function also records the pertinent information in date stamped text files whose names include matlab. The Matlab only variant is capable of having a measurement matrix and vector passed, but in that case the algorithms should be directly called as the Matlab function from the directory GAGA/gaga_matlab/algorithms.

4 Subroutine Timings

In this section we present least squares fits to average timings of the subroutines from Sec. 2.1 and the random problem generation described in Sec. 2.2. Rather than record the timings of each subroutine separated from their intended use in the greedy Algorithms 1 to 6, the timings we present are recorded from implementations of Alg. 4 and 5, which are sufficient to test each subroutine. Moreover, for conciseness and to ensure that all aspects of the algorithms tasks are included, we often report timings

Admissible options	
variable name	value type and default
'tol'	specifying convergence rate stopping conditions, positive float, default 0.001.
'maxiter'	maximum number of iterations, positive integer, default 300 for HTP and CSMPSP, 5000 for other algorithms.
'vecDistribution'	distribution of the nonzeros in the sparse vector for the test problem instance, string options: 'binary' for ± 1 with equal probability (default), 'gaussian' for Normal $\mathcal{N}(0,1)$, and 'uniform' for uniform from the interval zero to one, $\mathcal{U}(0,1)$.
'matrixEnsemble'	distribution of the nonzeros in the measurement matrix (not used for *dct*) for the test problem, string options: 'binary' for $\pm 1/\sqrt{p}$ with equal probability (default for *smv*), 'gaussian' for Normal $\mathcal{N}(0,1)$ (only valid for *gen*, default for *gen*), and 'ones' for all nonzeros all equal to one (only valid for *smv*).
'seed'	seed for random number generator, unsigned int, default `clock()`.
'numBins'	number of bins to use for order statistics, positive integer, default to $\max(n/20, 1000)$
'kFixed'	flag to force the k used in the problem generate to be that specified, string options: 'off' (default) and 'on'.
'noise'	level of additive normally distributed noise as a fraction of the $\|Ax\|_2$, non-negative float, default to 0.
'convRateNum'	number of the last iterations to use when calculating average convergence rate, positive integer, default 16.
'gpuNumber'	identify GPU to be used, non-negative integer, default to 0.
'threadsPerBlockn'	number of threads per block for kernels acting on vectors of length n, positive integer, default to $\min(n, max_threads_per_block)$.
'threadsPerBlockm'	number of threads per block for kernels acting on vectors of length m, positive integer, default to $\min(m, max_threads_per_block)$.
'threadsPerBlocknp'	number of threads per block for kernels acting on vectors of length $n \cdot p$ (for *smv*), positive integer, default to $\min(np, max_threads_per_block)$.
'threadsPerBlockBin'	number of threads per block for kernels acting on vectors of length *numBin*, positive integer, default to $\min(numBins, max_threads_per_block)$.
'timing'	indicates that times per iteration should be recorded, string options: 'off' (default) and 'on'.
'alpha'	specifying fraction $(1-\alpha)$ of bins counted in early support set identification steps, float between $(0,1)$, default to 0.25 which counts 75% of the bins initially. (only valid with 'timing' set to 'on'.).
'supportFlag'	method by which the support set is identified, integer options (only valid with 'timing' set to 'on'): 0 (default) for dynamic binning where binning is conducted only when the support set could have changed, 1 for binning at every iteration, 2 for using `thrust::sort` to find the largest entries when the support set could have changed, and 3 for using `thrust::sort` at every iteration.

Table 4 Optional arguments for `gaga_cs` set using gagaOptions.

not of single subroutines, but instead combinations of subroutines that compose Algorithms 1 to 6. For instance, in Section 4.1 we report the timings of `RestrictedSD` as implemented in NIHT along with the time to update the stopping criteria. Timings are presented for the matrix ensembles and problem size n listed in Tab. 5, with the n selected based upon memory constraints and the time taken to complete the tests (just over five days for the data presented here). The values of p tested for *smv* are selected to balance speed and efficacy of the algorithms [6]. All tests are conducted for randomly generated input vectors as described in Section 2.2.1 with the random sign pattern $\{-1,1\}$ distribution.

mat	*n* tested	Matrix generation
gen	$2^{10}, 2^{12}, 2^{14}$	$\mathcal{N}(0, m^{-1})$
dct	2^{10+2j} for $j = 0, 1, \ldots, 5$	Uniform at random
smv	2^{10+2j} for $j = 0, 1, \ldots, 4$	Uniform $\pm p^{-1/2}$ for GPU $p = 3, \ldots, 7$ and CPU $p = 4, 7$

Table 5 Matrix ensembles tested with list of large problem size n.

For each n listed in Tab. 5, tests are conducted for each value of $m = \lceil n\delta \rceil$ and each δ from (3) that results in $m \geq 100$.

$$\delta_{list} = 10^{-3} \cup \{2j * 10^{-3}\}_{j=1}^{5} \cup \{2j * 10^{-2}\}_{j=1}^{5} \cup \left\{0.1 + j\frac{0.89}{19}\right\}_{j=1}^{19} \tag{3}$$

For each (m, n) pair, tests are conducted with the GPU implementation for each independent value of $k = \lceil jm/49 \rceil$ starting with $j = 1$ and increasing by one until NIHT or HTP with one of the matrix ensembles in Tab. 5 fails to recover the measured vector within an ℓ_∞ error of 10^{-3} in each of ten consecutive attempts. Tests are conducted similarly for the CPU implementation but for $k = \lceil (2j-1)m/49 \rceil$ and are terminated after five consecutive failures recovering the measured vector.

NIHT and HTP are tested with the same stopping criteria, terminating when one of the following is achieved:

1. Scaled convergence $\|y - Ax_l\|_2 \leq 10^{-3}\frac{m}{n}$,
2. Divergence $\|y - Ax_l\|_2 > 100\|y - Ax_0\|_2$,
3. Convergence to an incorrect solution or small additive change
$$\max_{j=0,\ldots 15} \left| \|y - Ax_{l-j}\|_2 - \|y - Ax_{l-j-1}\|_2 \right| < 10^{-6}$$
4. Convergence to an incorrect solution or slow geometric convergence rate
$$\left(\frac{\|y - Ax_l\|_2}{\|y - Ax_{l-15}\|_2} \right)^{1/15} > 0.999$$
if $l > 750$ for NIHT and $l > 125$ for HTP.
5. Number of iterations exceeds 5000 for NIHT or 300 for HTP.

For each n in Tab. 5, NIHT and HTP are applied for many (k, m, n) as described in Section 4, with random problem instances generated as described in 2.2.1. The random problem generation does not ensure that the measured vector has the requested sparsity k, though typically it is very close. Each (k, m, n) tested is associated with the value of k from $\lceil jm/49 \rceil$ that is nearest, and for each of these proxy (k, m, n) all recorded timings are included in lists to be used for the reported values in the subsequent sections. Scripts are included with the software to produce similar data and presented tables and plots in the following sections.

All tests were conducted on a GPU workstation including dual Intel Xeon X5650 CPUs with 24GB of 1333MHz RAM and a single NVIDIA c2050 GPU. The CUDA

software was compiled using the 4.0 compiler [31] and all MatLab tests used version R2011b with automatic multi-threading. Average time-per-iteration values for `RestrictedSD` are presented in Section 4.1 with values recorded including the times for NIHT iteration lines 1 and 4 combined. Average times for `Threshold` combined with either `FindSupport` or `FindSupport_sort`, with values recorded from NIHT iteration lines 2 and 3 combined, are presented in Section 4.2. Average times for the projection portion of `RCGProjection` divided by the number of CG iterations conducted, to approximate the average time of `RestrictedCG`, with values recorded from the interior of HTP iteration line 4 are presented in Section 4.3. Average times for random problem generation and all initial memory allocations are presented in Section 4.4 with the values recorded coming from the above mentioned testing of NIHT. In each of the above tests, times are presented for both the GPU and CPU variants. The multiplicative increase in the median speed of the GPU timings as compared with the CPU timings are presented in Section 4.5.

4.1 `RestrictedSD`

The timings presented here are the combination of NIHT iteration lines 1 and 4, which include both a call to `RestrictedSD` and an update to the stopping criteria conditions which are calculated on the CPU. `RestrictedSD` includes two applications of the matrix-vector product A and one application of A^*, in addition to thresholding and vector operations. For each (k,m,n) tested as described in Section 4, the average of the recorded times for the combination of NIHT iteration lines 1 and 4 is evaluated and used to construct least squares fits of the average time per iteration. Least squares fits of the average times are calculated for each of four models: a constant time, constant plus linear in δ, constant plus linear in ρ, and constant plus linear in δ and ρ, which we refer to as the bilinear model. For conciseness we report values for only one of the models. If the least squares error for the bilinear model is less than 70% of the smaller of the two least squares errors of the linear models then we report the bilinear model coefficients; if this is not the case and one of the linear models has a least squares error that is less than 90% of the constant model then we report the coefficients of the more accurate of the linear models; otherwise we report the constant model coefficient. Scripts are included with the software that both replicate the below presented tables and which generate full tables with each of the four least squares models. All times presented are reported in *milliseconds*.

The least squares models for the *dct* matrix ensemble are presented in Tab. 6. The GPU implementation has an approximate minimum time of half a millisecond for $n = 2^{10}$. The time increases with n, exhibiting a δ dependence for $n \geq 2^{14}$, and the largest time for $n = 2^{20}$ and $\delta = 0.99$ showing a near four-fold increase in the time as compared to $n = 2^{18}$. The largest time for the GPU implementation is just over $10ms$. The CPU implementation similarly shows a near constant minimum time of approximately $33ms$ for smaller values of n, and again exhibits a linear dependence in δ for the largest value of n.

The least squares models for the *gen* matrix ensemble are presented in Tab. 7. Both the GPU and CPU implementations show a strong linear dependence on δ with

Table 6 Least squares model $Const. + \alpha\delta$ for the average time in milliseconds of NIHT iteration lines 1 and 4 using the *dct* matrix ensemble. This time includes both a call to `RestrictedSD` as well as updating the stopping criteria.

	n	Const.	α	ℓ_∞ error
gpu	2^{10}	0.545	-	0.113
	2^{12}	0.577	-	0.106
	2^{14}	0.648	0.048	0.0846
	2^{16}	0.861	0.167	0.0798
	2^{18}	1.87	0.816	0.0775
	2^{20}	6.05	4.68	0.378
cpu	2^{10}	33.9	-	6.39
	2^{12}	36	-	7.68
	2^{14}	39.5	-	8.2
	2^{16}	57.9	-	8.24
	2^{18}	119	15.5	10.8
	2^{20}	490	-	324

a small constant component, and have rapid increases with n. In particular, for $n = 2^{14}$ the *gen* ensemble can take approximately $43ms$ whereas corresponding tests for the *dct* ensemble take under $0.7ms$. For the smallest values $n = 2^{10}$ the CPU implementation is roughly as fast as the GPU implementation, but for $n = 2^{14}$ the GPU implementation is approximately sixteen times faster than the CPU implementation; further comparisons are presented in Section 4.5.

Table 7 Least squares model $Const. + \alpha\delta$ for the average time in milliseconds of NIHT iteration lines 1 and 4 using the *gen* matrix ensemble. This time includes both a call to `RestrictedSD` as well as updating the stopping criteria.

	n	Const.	α	ℓ_∞ error
gpu	2^{10}	0.765	-	0.0823
	2^{12}	0.666	2.59	0.0922
	2^{14}	0.625	42.8	1.44
cpu	2^{10}	0.227	1.21	1.08
	2^{12}	-0.00994	42.9	10.3
	2^{14}	-22	860	163

The least squares models for the *smv* matrix ensemble are presented in Tab. 8. The GPU implementation has an approximate minimum time of $0.62ms$ for $n = 2^{10}$. The GPU implementations exhibit a ρ dependence for larger n, which is increasingly dominant for larger values of the number of nonzeros per column, p. The GPU implementation also exhibits a less pronounced δ dependence for the largest values of n. In contrast, the CPU implementation has a substantial dependence on δ and essentially no ρ dependence. The GPU implementation shows a slow growth but consistent increase in the time as p is increased, and begins to exhibit a linear increase with n for the largest values of n. The CPU variant appears to have a superlinear increase with n. For small problem sizes the *smv* ensemble has times similar to the *dct* ensemble, with *dct* taking nearly twice as long for the largest values of n.

Table 8 Least squares model $Const. + \alpha\delta + \beta\rho$ for the average time in milliseconds of NIHT iteration lines 1 and 4 using the *smv* matrix ensemble. This time includes both a call to `RestrictedSD` as well as updating the stopping criteria.

	n	Const.	α	β	ℓ_∞ error
gpu p=3	2^{10}	0.624	-	-	0.102
	2^{12}	0.654	-	-	0.107
	2^{14}	0.674	-	0.285	0.103
	2^{16}	0.81	-	0.851	0.16
	2^{18}	1.3	0.613	1.78	0.418
gpu p=4	2^{10}	0.622	-	-	0.0807
	2^{12}	0.66	-	-	0.0872
	2^{14}	0.696	-	0.348	0.082
	2^{16}	0.878	0.137	0.548	0.106
	2^{18}	1.68	0.586	1.83	0.173
cpu p=4	2^{10}	0.238	-	-	1.07
	2^{12}	0.431	-	-0.532	0.482
	2^{14}	0.93	0.231	-	0.0896
	2^{16}	3.42	1.72	-	0.678
	2^{18}	17.5	9.9	-	2.51
gpu p=5	2^{10}	0.624	-	-	0.244
	2^{12}	0.648	-	0.181	0.0913
	2^{14}	0.71	-	0.431	0.109
	2^{16}	0.928	0.162	0.676	0.13
	2^{18}	1.92	0.663	2.25	0.157
gpu p=6	2^{10}	0.624	-	-	0.128
	2^{12}	0.652	-	0.218	0.085
	2^{14}	0.733	-	0.478	0.0812
	2^{16}	1.01	0.155	0.783	0.111
	2^{18}	2.21	0.693	2.66	0.207
gpu p=7	2^{10}	0.627	-	-	0.127
	2^{12}	0.657	-	0.261	0.186
	2^{14}	0.753	-	0.553	0.0946
	2^{16}	1.15	-	1.25	0.145
	2^{18}	2.5	0.741	3.14	0.25
cpu p=7	2^{10}	0.306	-	-0.356	0.144
	2^{12}	0.506	-	-0.307	0.14
	2^{14}	1.3	0.437	-	0.407
	2^{16}	4.98	3.55	-	0.839
	2^{18}	25.8	17	-	5.48

4.2 Support set identification via `FindSupport` and `FindSupport_sort`

Whereas the computational time of `RestrictedSD` and `RestrictedCG` are dominated by the matrix-vector multiplications with predictable dependence on δ and ρ, `FindSupport` has a far less predictable behavior and is not well represented by simple linear regressions. Rather, the timings recorded for `FindSupport` are presented as plots. For conciseness we present only the plots for the *dct* ensemble using the GPU implementation of NIHT. Scripts are included with the software that both replicate the below presented plots and generate plots for the *smv* and *gen* ensembles as well as HTP and CPU implementations.

`FindSupport` differs from the more standard `FindSupport_sort` in two ways: rather than sorting the vector it only identifies the support set needed for `Threshold`, and `FindSupport` only computes a new support set if the update to the vector is sufficiently large that the support set could have changed. Both of these features result in reduced execution times. Fig. 1 - 3 show average times for the proxy (k,m,n) as described in Section 4.1 for $n = 2^{16}, 2^{18}$, and 2^{20}. In each plot: Panel (a) is `FindSupport`, Panel (b) uses `FindSupport` with *minValue* set to *maxValue* to force the support set identification at each call, Panel (c) uses sorting but only when the update is sufficiently large that the support set could have changed, Panel (d) uses sorting at each iteration, Panel (e) is the ratio of the times in Panel (a) over those in Panel (b), and Panel (f) is the ratio of the times in Panel (a) over those in Panel (c).

Contrasting Panels (d) and (c) as well as (b) and (a) shows the reduction in time achieved by only applying the support set identification when the support set could have changed. Panel (e) show the gain for only identifying the support set when needed. Panel (f) show the gain for using the linear binning approach of `FindSupport` as opposed to sorting. Fig. 3 shows that for $n = 2^{20}$ the sorting at each iteration takes approximately 2.5*ms* but `FindSupport` takes between 0.3*ms* and 0.6*ms*. In contrast, Tab. 6 shows that for the same problem size `RestrictedSD` takes between 6*ms* and 10*ms*. For *dct* and $n = 2^{20}$ sorting at each iteration can take between 25% and 40% of the time per iteration, whereas `FindSupport` takes between 5% and 10%. Though a modest reduction in time, the result is the time per iteration of the GPU implementation of the greedy algorithms being almost exclusively the time for the necessary matrix-vector multiplications.

4.3 `RestrictedCG`

The timings presented here are the projection portion of `RCGProjection` divided by the number of CG iterations conducted with values recorded from the interior of HTP iteration line 4. Each average includes both a call to `RestrictedCG` and an update to the CG stopping criteria, which is calculated on the CPU, similar to the NIHT stopping criteria. `RestrictedCG` includes one application of the matrix-vector product A and one application of A^* in addition to thresholding and vector operations, which is approximately the same computational cost as that of `RestrictedSD` presented in Section 4.1. The values of (k,m,n) tested and least squares model fits are calculated and described in Section 4.1.

The least squares models for the *dct* matrix ensemble are presented in Tab. 9. The GPU implementation has an approximate minimum time of 0.35*ms* for $n = 2^{10}$ which is approximately two thirds of the timings presented in Tab. 6. As observed for `RestrictedSD` in Section 4.1, the time increases with n, exhibiting a δ dependence for $n \geq 2^{14}$, and the largest time for $n = 2^{20}$ and $\delta = 0.99$ showing a near four-fold

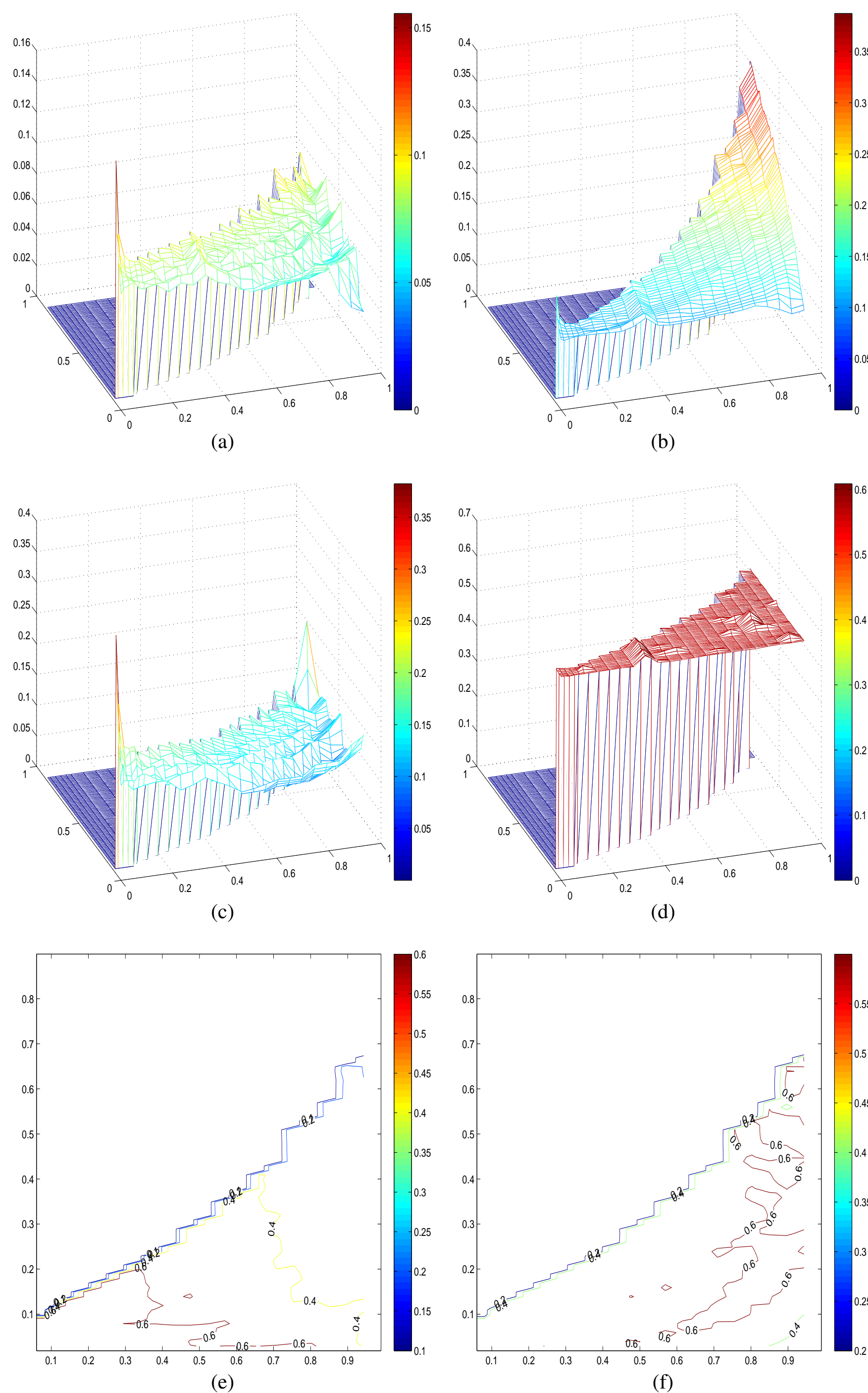

Fig. 1 Plot of average time (in milliseconds) for support set detection per iteration with $n = 2^{16}$. The GPU implementation of NIHT with the *dct* matrix ensemble. Panel (a) is `FindSupport`. Panel (b) uses `FindSupport` with minValue set to maxValue to force the support set identification at each call. Panel (c) uses sorting but only when the update is sufficiently large that the support set could have changed. Panel (d) uses sorting at each iteration. Panel (e) is the ratio of the times in Panel (a) over those in Panel (b). Panel (f) is the ratio of the times in Panel (a) over those in Panel (c).

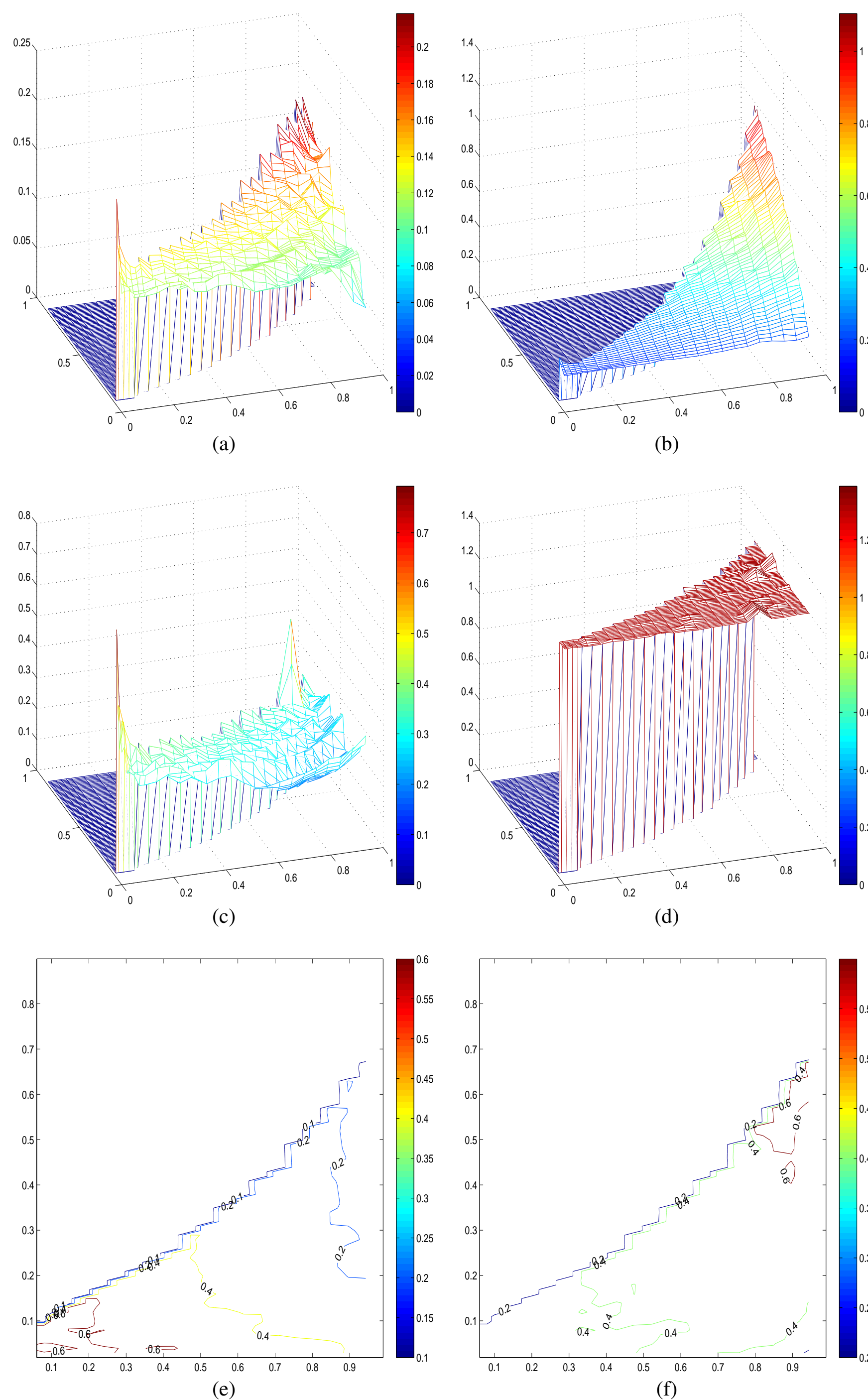

Fig. 2 Plot of average time (in milliseconds) for support set detection per iteration with $n = 2^{18}$. The GPU implementation of NIHT with the *dct* matrix ensemble. Panel (a) is `FindSupport`. Panel (b) uses `FindSupport` with minValue set to maxValue to force the support set identification at each call. Panel (c) uses sorting but only when the update is sufficiently large that the support set could have changed. Panel (d) uses sorting at each iteration. Panel (e) is the ratio of the times in Panel (a) over those in Panel (b). Panel (f) is the ratio of the times in Panel (a) over those in Panel (c).

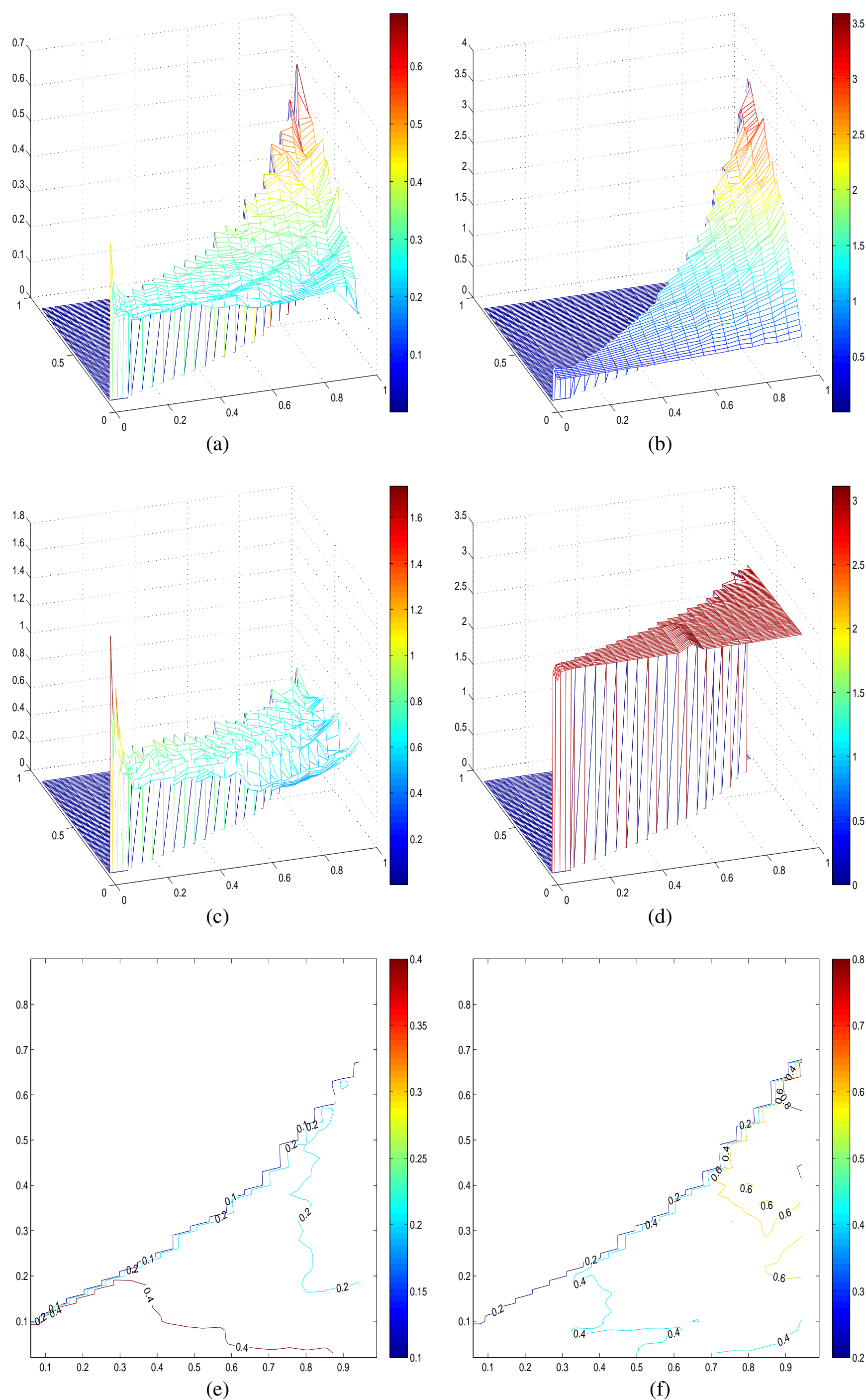

Fig. 3 Plot of average time (in milliseconds) for support set detection per iteration with $n = 2^{20}$. The GPU implementation of NIHT with the *dct* matrix ensemble. Panel (a) is `FindSupport`. Panel (b) uses `FindSupport` with minValue set to maxValue to force the support set identification at each call. Panel (c) uses sorting but only when the update is sufficiently large that the support set could have changed. Panel (d) uses sorting at each iteration. Panel (e) is the ratio of the times in Panel (a) over those in Panel (b). Panel (f) is the ratio of the times in Panel (a) over those in Panel (c).

increase in the time as compared to $n = 2^{18}$. The largest time for the GPU implementation is approximately $7.5ms$. The CPU implementation similarly shows a near constant minimum time of approximately $25ms$ for smaller values of n, and again exhibits a linear delta dependence in δ for the largest value of n.

Table 9 Least squares model $Const. + \alpha\delta$ for the average time in milliseconds of the projection portion of `RCGProjection` divided by the number of CG iterations conduced, to approximate the average time of `RestrictedCG`, with values recorded from the interior of HTP iteration line 4 for the *dct* matrix ensemble.

	n	Const.	α	ℓ_∞ error
gpu	2^{10}	0.346	-	0.0643
	2^{12}	0.37	-	0.0734
	2^{14}	0.417	0.0179	0.0508
	2^{16}	0.562	0.0833	0.0523
	2^{18}	1.29	0.519	0.0548
	2^{20}	4.14	3.23	0.102
cpu	2^{10}	24.2	-	2.02
	2^{12}	24.6	-	5.05
	2^{14}	28.7	-	6.9
	2^{16}	39.7	-	4.96
	2^{18}	79.7	9.5	9.75
	2^{20}	351	36.1	80.6

The least squares models for the *gen* matrix ensemble are presented in Tab. 10. Both the GPU and CPU implementations show a strong linear dependence on δ with a small constant component, and have rapid increases with n. For the largest value of $n = 2^{14}$ the GPU implementation is nearly sixteen times faster than the CPU implementation. Both ensembles exhibit a rapid rise in the execution time as n increases.

Table 10 Least squares model $Const. + \alpha\delta + \beta\rho$ for the average time in milliseconds of the projection portion of `RCGProjection` divided by the number of CG iterations conduced, to approximate the average time of `RestrictedCG`, with values recorded from the interior of HTP iteration line 4 for the *gen* matrix ensemble.

	n	Const.	α	β	ℓ_∞ error
gpu	2^{10}	0.474	-0.0407	-	0.0522
	2^{12}	0.383	1.71	-	0.0659
	2^{14}	0.372	28.2	-	0.712
cpu	2^{10}	0.387	0.77	-0.822	0.274
	2^{12}	-0.422	28.2	-	4.67
	2^{14}	16.5	449	-	155

The least squares models for the *smv* matrix ensemble are presented in Tab. 11. The GPU implementation has an approximate minimum time of $0.36ms$ for $n = 2^{10}$. The GPU implementations exhibit a ρ dependence for larger n and p; though less pronounced than the ρ dependence in Tab. 8. The GPU implementation exhibits a δ dependence for the largest value of n, with this dependence decreasing for larger val-

Table 11 Least squares model $Const. + \alpha\delta + \beta\rho$ for the average time in milliseconds of the projection portion of `RCGProjection` divided by the number of CG iterations conduced, to approximate the average time of `RestrictedCG`, with values recorded from the interior of HTP iteration line 4 for the *smv* matrix ensemble.

	n	Const.	α	β	ℓ_∞ error
gpu p=3	2^{10}	0.355	-	-	0.0994
	2^{12}	0.367	-	-	0.0626
	2^{14}	0.401	-	-	0.174
	2^{16}	0.612	-	-0.379	0.464
	2^{18}	1.19	-	-	0.906
gpu p=4	2^{10}	0.361	-	-	0.0508
	2^{12}	0.384	-	-	0.138
	2^{14}	0.411	0.029	-	0.0629
	2^{16}	0.565	0.13	-	0.095
	2^{18}	1.27	0.507	-	0.343
cpu p=4	2^{10}	0.231	-	-0.18	0.071
	2^{12}	0.391	-	-0.561	0.243
	2^{14}	0.812	-	-	0.155
	2^{16}	2.66	0.768	-	0.411
	2^{18}	12.6	6.04	-	1.32
gpu p=5	2^{10}	0.363	-	-	0.0485
	2^{12}	0.391	-	-	0.0868
	2^{14}	0.433	-	0.168	0.0551
	2^{16}	0.626	0.152	-	0.112
	2^{18}	1.52	0.587	-	0.398
gpu p=6	2^{10}	0.365	-	-	0.0877
	2^{12}	0.398	-	-	0.0665
	2^{14}	0.454	-	0.197	0.125
	2^{16}	0.729	-	0.576	0.116
	2^{18}	1.77	0.445	1.16	0.337
gpu p=7	2^{10}	0.364	-	-	0.056
	2^{12}	0.391	0.0259	-	0.0529
	2^{14}	0.473	-	0.266	0.0889
	2^{16}	0.804	-	0.685	0.254
	2^{18}	2.07	0.411	1.57	0.4
cpu p=7	2^{10}	0.279	-	-0.294	0.258
	2^{12}	0.427	-	-0.381	0.134
	2^{14}	1.08	-	-	1.4
	2^{16}	3.68	1.6	-	1.15
	2^{18}	17.5	9.37	-	2.51

ues of p. The CPU implementation has essentially no ρ dependence, and a significant dependence on δ. Behavior with p and n are consistent with Tab. 8.

4.4 Problem generation and memory allocation

One of the primary motivations for the development of this software is to allow large-scale testing of the growing number of competing algorithms for compressed sensing. Such large-scale testing includes both evaluation of algorithm performance for large problem sizes n, as well as testing the algorithms on a large number of random prob-

lem instances. Results of such testing are presented in [6]. Efficient testing of large numbers of problems requires the ability to generate random problems with the problem generation time small compared to time needed for the algorithm to terminate.

The timings presented here are the time to generate a random problem and all memory allocation as well as creating the initial step by computing A^*y, finding its k largest entries using `FindSupport` and thresholding to the corresponding support set using `Threshold`. The values of (k,m,n) tested and least square model fits are calculated at described in Section 4.1.

The least squares models for the *dct* matrix ensemble are presented in Tab. 12. The GPU implementation has an approximate nearly constant minimum time of 19*ms* for $n \leq 2^{16}$, increasing up to about 60*ms* for $n = 2^{20}$ where a small linear dependence in δ is also observed, which is consistent with `RestrictedSD`. The CPU implementation similarly shows a near constant minimum time of approximately 14*ms* for smaller values of n, with an earlier significant δ dependence as n increases, with the total time for $n = 2^{18}$ approaching 100*ms*.

Table 12 Least squares model $Const. + \alpha\delta + \beta\rho$ for the average time in milliseconds for random problem generation and memory allocation for NIHT using the *dct* matrix ensemble.

	n	Const.	α	β	ℓ_∞ error
gpu	2^{10}	18.7	-0.254	-	0.353
	2^{12}	18.7	-	-	0.82
	2^{14}	19.1	-	-	0.432
	2^{16}	20.7	0.393	-	1.05
	2^{18}	27	2.64	-	1.63
	2^{20}	49.6	6.38	-	3.73
cpu	2^{10}	13	-	-	2.66
	2^{12}	14.9	-	-	2.98
	2^{14}	19.3	-	-	4.72
	2^{16}	31.8	8.38	-	44.3
	2^{18}	88	11.1	-	13.8
	2^{20}	294	-	67.4	214

The least squares models for the *gen* matrix ensemble are presented in Tab. 13. Both the GPU and CPU implementations show a pronounced linear dependence on δ. The GPU implementation time increases from a nearly constant time of about 37*ms* for $n = 2^{10}$ to between 35*ms* and 560*ms* for $n = 2^{14}$. The CPU implementation initially has a smaller time for $n = 2^{10}$ of between 3*ms* and 23*ms*, but increases dramatically when $n = 2^{14}$ to between 10*ms* for small δ to $3,800$*ms* for δ near one.

The least squares models for the *smv* matrix ensemble are presented in Tab. 14. The GPU implementation has a nearly constant time of 19*ms* for $n \leq 2^{16}$, increasing only modestly to between 26*ms* and 29*ms* for $n = 2^{18}$. The GPU implementation shows no ρ dependence with the exception of one apparent anomaly, and shows a small δ dependence only for $n = 2^{18}$ and $p \geq 4$. Increasing p from 3 to 7 introduces only a modest (less than 2%) increase in the time. The CPU implementation exhibits a strong δ dependence for $n \leq 2^{16}$ and dramatic increase with n from about 33*ms* for

Table 13 Least squares model $Const. + \alpha\delta$ for the average time in milliseconds for random problem generation and memory allocation for NIHT using the *gen* matrix ensemble.

	n	Const.	α	ℓ_∞ error
gpu	2^{10}	36.4	6.87	0.825
	2^{12}	36	34	1.17
	2^{14}	35	529	5.81
cpu	2^{10}	3.12	19.6	5.35
	2^{12}	9.55	248	12.1
	2^{14}	10.1	3.84e+03	90.1

$n = 2^{10}$ to $43,000ms$ for $n = 2^{18}$. The CPU implementation shows a small increase in time with p.

4.5 GPU Acceleration Ratio

For each of the tests described in Sections 4.1–4.3 there are (k,m,n) triples for which both the GPU and CPU timings are recorded. This section gives a rough calculation of the general acceleration of the GPU implementation as compared to the CPU implementation. For each n where GPU and CPU timings are available we compute an acceleration factor as follows: each associated proxy (k,m) as described in Section 4.1 that was tested for both GPU and CPU has the median of the GPU and CPU times calculated and the ratio of the GPU over CPU times calculated. After each (k,m) acceleration ratio is calculated, for a given n, we report the average of the acceleration ratios. For NIHT this process is conducted for each of `RestrictedSD`, `FindSupport`, and the problem generation, and HTP also includes `RestrictedCG`. The average of the median acceleration ratios for NIHT are shown in Tab. 15 and for HTP are shown in Tab. 16.

`RestrictedSD` is observed to be between 54 and 77 times faster for the *dct* ensemble, as much as ten times faster for the *smv* ensemble for $n = 2^{18}$ and as much as sixteen times faster for the *gen* ensemble for $n = 2^{14}$. `FindSupport` is between 28 and 60 times faster for the *dct* ensemble, as much as 70 times faster for the *smv* ensemble for $n = 2^{18}$ reducing to equally fast for $n = 2^{10}$, and as much as sixteen times faster for the *gen* ensemble with $n = 2^{14}$. The random problem generation for the *dct* ensemble is moderately slower for $n < 2^{14}$, increasing to about six times faster for $n = 2^{20}$, but is dramatically (over one-thousand times) faster for the *smv* ensemble for n large, and as much as six times faster for the *gen* ensemble with $n = 2^14$. The acceleration of the random problem generation for large problem sizes is an essential part of large-scale testing where these times often dominate those of the algorithm for all but the (k,m,n) nearest to the transition where the algorithm is unable to recover the measured signal. `RestrictedCG` has acceleration ratios similar to those of `RestrictedSD`.

Table 14 Least squares model $Const. + \alpha\delta + \beta\rho$ for the average time in milliseconds for random problem generation and memory allocation for NIHT using the *smv* matrix ensemble.

	n	Const.	α	β	ℓ_∞ error
gpu p=3	2^{10}	18.9	-0.275	-	0.683
	2^{12}	18.8	-	-	0.715
	2^{14}	19.2	-	-	1.04
	2^{16}	20.6	-	0.829	1.05
	2^{18}	26.5	-	-	1.85
gpu p=4	2^{10}	18.7	-	-	0.569
	2^{12}	18.8	-	-	0.604
	2^{14}	19.3	-0.271	-	0.516
	2^{16}	20.8	-	-	0.841
	2^{18}	26.3	0.849	-	0.988
cpu p=4	2^{10}	37	78.9	-	11.3
	2^{12}	332	536	-	195
	2^{14}	1.85e+03	1.35e+03	-	1.29e+03
	2^{16}	8.74e+03	3.57e+03	-	5.83e+03
	2^{18}	3.79e+04	9.99e+03	-	2.62e+04
gpu p=5	2^{10}	18.8	-	-	0.96
	2^{12}	18.6	-	-	0.676
	2^{14}	19.5	-0.222	-	0.386
	2^{16}	21	-	-	0.932
	2^{18}	26.8	0.867	-	0.865
gpu p=6	2^{10}	18.8	-	-	0.673
	2^{12}	18.7	-	-	0.627
	2^{14}	19.6	-0.243	-	0.459
	2^{16}	21.1	-	-	1.03
	2^{18}	27.7	0.696	-	1.11
gpu p=7	2^{10}	18.8	-	-	0.452
	2^{12}	18.7	0.256	-	0.574
	2^{14}	19.6	-	-	0.478
	2^{16}	21.4	-	-	0.624
	2^{18}	28.8	0.519	-	0.827
cpu p=7	2^{10}	33.8	83.4	-	7.9
	2^{12}	325	541	-	205
	2^{14}	1.95e+03	1.28e+03	-	1.38e+03
	2^{16}	8.9e+03	3.69e+03	-	5.97e+03
	2^{18}	4.41e+04	-	-	3.21e+04

5 Conclusions and Future Work

This software package allows for the timely testing of algorithms of both large problem sizes as well as testing large numbers of randomly generated problems. Both GPU implementations, written in CUDA, and CPU implementations, written in MatLab, are included to permit evaluation of the acceleration achieved by the GPU implementation compared to the CPU implementation; furthermore, users who wish to add functionality to the GPU implementation but are less familiar with CUDA have a more familiar MatLab implementation for development.

For the *dct* matrix ensemble, the included greedy algorithms have a very small time per iteration due to their extremely efficient GPU implementation of the mat-

Table 15 Multiplicative acceleration factor for NIHT of median times for the GPU over CPU times.

	n	p	RestrictedSD	FindSupport	Prob. Gen.
dct	2^{10}		62.64	42.54	0.70
	2^{12}		63.09	42.44	0.80
	2^{14}		63.21	42.16	1.04
	2^{16}		64.46	41.59	1.77
	2^{18}		54.11	38.45	3.20
	2^{20}		57.94	38.82	5.80
smv	2^{10}	4	0.33	1.85	4.26
	2^{12}	4	0.52	4.10	32.32
	2^{14}	4	1.41	14.64	135.08
	2^{16}	4	4.29	43.04	521.60
	2^{18}	4	10.43	71.50	1630.08
	2^{10}	7	0.30	2.16	4.26
	2^{12}	7	0.63	3.48	33.92
	2^{14}	7	1.86	12.86	142.53
	2^{16}	7	5.42	37.11	526.82
	2^{18}	7	10.80	55.60	1556.44
gen	2^{10}		1.06	2.07	0.34
	2^{12}		10.36	4.09	2.53
	2^{14}		16.75	6.17	5.85

Table 16 Multiplicative acceleration factor for HTP of median times for the gpu over cpu times.

	n	p	RestrictedSD	FindSupport	Prob. Gen.	RestrictedCG
dct	2^{10}		76.88	29.96	0.75	70.20
	2^{12}		72.42	28.21	0.80	69.00
	2^{14}		69.94	33.82	1.05	68.27
	2^{16}		65.11	33.79	1.70	66.09
	2^{18}		54.17	37.92	2.96	53.30
	2^{20}		61.47	60.58	6.40	61.40
smv	2^{10}	4	0.39	0.95	4.26	0.51
	2^{12}	4	0.61	2.16	31.34	0.79
	2^{14}	4	1.57	4.32	132.79	1.90
	2^{16}	4	4.41	13.09	504.11	4.74
	2^{18}	4	10.01	24.89	1650.11	9.97
	2^{10}	7	0.42	0.93	4.26	0.52
	2^{12}	7	0.72	1.55	32.60	0.88
	2^{14}	7	1.99	4.63	138.78	2.14
	2^{16}	7	5.38	18.77	520.16	5.25
	2^{18}	7	9.89	38.89	1550.09	9.24
gen	2^{10}		1.28	1.19	0.35	1.41
	2^{12}		10.34	2.16	2.55	10.47
	2^{14}		15.71	4.94	5.71	15.43

vecs, Tab. 6, which are between 50 and 70 times faster than their CPU implementation. In addition, the *dct* ensemble benefits from a small time for the random problem generation due to the few random numbers needed. The included greedy algorithms are observed to have nearly the same recovery ability for the *dct* and *smv* matrix ensembles for $\delta < 1/5$ and $p = 7$ [6]. In this regime, the *smv* is observed to have a time

per iteration similar the dct. The *smv* ensemble random problem generation requires a large number of random variables, which becomes prohibitively time consuming for the CPU implementation and n large, often with the problem generation taking many times longer than the time needed for the greedy algorithm to converge. Fortunately, the GPU implementation of the random problem generation is dramatically faster than the CPU implementation, allowing for efficient testing of large numbers of *smv* problems. Moreover, the *smv* ensemble is observed to be approximately ten fold faster for `RestrictedSD` with $n = 2^{18}$ on the GPU as compared to the CPU implementation, Tab. 15 and 16. The *gen* ensemble shows over ten fold improvement for `RestrictedSD` and `RestrictedCG` with $n = 2^{12}$ and $n = 2^{14}$ as well as a significant reduction in the time to generate random problems.

The first generation of the software presented here includes representative examples of greedy algorithms for compressed sensing, but is in no way exhaustive. Numerous competitive algorithms have not been included. The software has been written in a way to ease the inclusion of other algorithms. Scripts have been included which can repeat the tests presented here, including automatic generation of tables. These scripts allow users to replicate the results here, or determine the behavior and efficiencies of the software on their system, in addition to easily testing newly included algorithms. The software is available online at [5], without charge for academic researchers.

Acknowledgements We thank Stephen Wright and Shangkyun Lee for allowing inclusion of their *dct* matrix vector product code [25]. We also thank Erik Opavsky and Emircan Uysaler, Grinnell College students, for their dense matrix-vector product which is the foundation of our *gen* matrix-vector product.

References

1. Alabi, T., Blanchard, J., Gordon, B., Steinbach, R.: Fast k-selection algorithms for graphics processing units. ACM J. Experimental Algorithmics **17**(2) (2012). Article 4.2, Pages 4.2:1-4.2:29
2. Allison, D., Noga, M.: Selection by distributive partitioning. Information Processing Letters **11**(1), 7 – 8 (1980)
3. Beliakov, G.: Parallel calculation of the median and order statistics on GPUs with application to robust regression. Computing Research Repository **abs/1104.2** (2011)
4. Berg, E.v.d., Friedlander, M.P.: Probing the Pareto frontier for basis pursuit solutions. SIAM Journal on Scientific Computing **31**(2), 890–912 (2008)
5. Blanchard, J.D., Tanner, J.: GAGA: GPU Accelerated Greedy Algorithms (2013). URL www.gaga4cs.org. Version 1.0.0
6. Blanchard, J.D., Tanner, J.: Performance comparisons of greedy algorithms in compressed sensing. Submitted, www.math.grinnell.edu/∼blanchaj/PCGACS_preprint.pdf (2013)
7. Blumensath, T., Davies, M.E.: Iterative hard thresholding for compressed sensing. Appl. Comput. Harmon. Anal. **27**(3), 265–274 (2009)
8. Blumensath, T., Davies, M.E.: Normalised iterative hard thresholding; guaranteed stability and performance. IEEE Selected Topics in Signal Processing **4**(2), 298–309 (2010)
9. Candès, E.J.: Compressive sampling. In: International Congress of Mathematicians. Vol. III, pp. 1433–1452. Eur. Math. Soc., Zürich (2006)
10. Candès, E.J., Romberg, J.K., Tao, T.: Stable signal recovery from incomplete and inaccurate measurements. Comm. Pure Appl. Math. **59**(8), 1207–1223 (2006)
11. Candès, E.J., Tao, T.: Decoding by linear programming. IEEE Trans. Inform. Theory **51**(12), 4203–4215 (2005)
12. Cevher, V.: An ALPS view of sparse recovery. In: Acoustics, Speech and Signal Processing (ICASSP), 2011 IEEE International Conference on, pp. 5808–5811. IEEE (2011)

13. Dai, W., Milenkovic, O.: Subspace pursuit for compressive sensing signal reconstruction. IEEE Trans. Inform. Theory **55**(5), 2230–2249 (2009)
14. Donoho, D.L.: Neighborly polytopes and sparse solution of underdetermined linear equations (2004). Technical Report, Department of Statistics, Stanford University
15. Donoho, D.L.: Compressed sensing. IEEE Trans. Inform. Theory **52**(4), 1289–1306 (2006)
16. Donoho, D.L.: For most large underdetermined systems of equations, the minimal l_1-norm near-solution approximates the sparsest near-solution. Comm. Pure Appl. Math. **59**(7), 907–934 (2006)
17. Donoho, D.L.: High-dimensional centrally symmetric polytopes with neighborliness proportional to dimension. Discrete Comput. Geom. **35**(4), 617–652 (2006)
18. Donoho, D.L., Tanner, J.: Sparse nonnegative solution of underdetermined linear equations by linear programming. Proc. Natl. Acad. Sci. USA **102**(27), 9446–9451 (electronic) (2005)
19. Donoho, D.L., Tsaig, Y.: Fast solution of l1 minimization problems when the solution may be sparse. IEEE Trans. Inform. Theory **54**(11), 4789–4812 (2008)
20. Donoho, D.L., Tsaig, Y., Drori, I., Stark, J.L.: Sparse solution of underdetermined linear equations by stagewise orthogonal matching pursuit. IEEE Trans. Inform. Theory **58**(2), 1094–1121 (2012)
21. Figueiredo, M.A.T., Nowak, R.D., Wright, S.J.: Gradient projection for sparse reconstruction: Application to compressed sensing and other inverse problems. IEEE Selected Topics in Signal Processing **1**(4), 586–597 (2007)
22. Foucart, S.: Hard thresholding pursuit: an algorithm for compressive sensing. SIAM Journal on Numerical Analysis **49**(6), 2543–2563 (2011)
23. Hoberock, J., Bell, N.: Thrust: A parallel template library (2010). URL http://www.meganewtons.com/. Version 1.3.0, `http://www.meganewtons.com/`
24. Kyrillidis, A., Cevher, V.: Recipes on hard thresholding methods. In: Computational Advances in Multi-Sensor Adaptive Processing (CAMSAP), 2011 4th IEEE International Workshop on, pp. 353–356. IEEE (2011)
25. Lee, S., Wright, S.J.: Implementing algorithms for signal and image reconstruction on graphical processing units (2008). URL http://pages.cs.wisc.edu/ swright/GPUreconstruction/gpu_image.pdf
26. Mallat, S., Zhang, Z.: Matching pursuits with time-frequency dictionaries. Signal Processing, IEEE Transactions on **41**(12), 3397 –3415 (1993)
27. Merrill, D., Grimshaw, A.: High performance and scalable radix sorting: A case study of implementing dynamic parallelism for GPU computing. Parallel Processing Letters **21**(02), 245–272 (2011)
28. Monroe, L., Wendelberger, J., Michalak, S.: Randomized selection on the GPU. In: Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics, HPG '11, pp. 89–98. ACM, New York, NY, USA (2011)
29. Natarajan, B.K.: Sparse approximate solutions to linear systems. SIAM J. Comput. **24**(2), 227–234 (1995)
30. Needell, D., Tropp, J.: CoSaMP: Iterative signal recovery from incomplete and inaccurate samples. Appl. Comp. Harm. Anal. **26**(3), 301–321 (2009)
31. NVIDIA: Cuda toolkit 4.0 (2011). `http://developer.nvidia.com/cuda-toolkit-40`
32. Tropp, J.A.: Greed is good: algorithmic results for sparse approximation. IEEE Trans. Inform. Theory **50**(10), 2231–2242 (2004)
33. Tropp, J.A., Gilbert, A.C.: Signal recovery from random measurements via orthogonal matching pursuit. IEEE Trans. Info. Theory **53**(12), 4655–4666 (2007)
34. Wright, S.J., Nowak, R.D., Figueiredo, M.A.T.: Sparse reconstruction by separable approximation. Proc. International Conference on Acoustics, Speech, and Signal Processing (2008)
35. Yin, W., Osher, S., Goldfarb, D., Darbon, J.: Bregman iterative algorithms for ℓ^1-minimization with applications to compressed sensing. SIAM Journal on Imaging Science **1**(1), 143–168 (2008)