Implementing Algorithms for Signal and Image Reconstruction on Graphical Processing Units

Sangkyun Lee and Stephen J. Wright

***Abstract*—Several highly effective algorithms that have been proposed recently for compressed sensing and image processing applications can be implemented efficiently on commodity graphical processing units (GPUs). The properties of algorithms and application that make for efficient GPU implementation are discussed, and computational results for several algorithms are presented that show large speedups over CPU implementations.**

***Index Terms*—Graphical processing units, compressed sensing, image denoising, image deblurring.**

I. Introduction

Parallel computation has long been recognized as a means of speeding up computationally intensive numerical computing tasks. A great variety of architectures has been developed over the years to support different types of parallel computation. With the current ubiquity of multicore architectures, parallel processing has become the dominant paradigm in computing.

Most personal computers also contain another type of parallel processing device: *graphical processing units* (GPUs). GPUs were developed for real-time rendering or 3D graphics, but are increasingly being adopted for massively parallel numerical computating. GPUs feature a many-processor architecture and high-bandwidth memory that maximize multithreading performance, along with faster arithmetic units than those on CPUs. Many recent studies have reported significant performance boosts by using GPUs to run critical numerical computing tasks in various applications, including pattern analysis [1], [2], biomedical imaging [3], DNA sequence alignment [4], molecular modeling and simulation [5], [6], multibody dynamics [7], and quantum chemistry [8].

Our focus in this paper is on GPU implementations of algorithms that have been proposed recently for reconstruction of sparse signals from random observations (compressed sensing) and for image denoising and deblurring. These problems typically involve a large number of unknowns but, unlike many problems in numerical computing, the amount of data needed to specify the problem typically is often less than the number of unknowns. The major computational operations that are used in the algorithms we described here — which are among the most effective algorithms available for these problems, even when implemented on CPUs — are readily implemented with high efficiency on GPUs.

This research was supported in part by NSF under Grants CCF-0430504 and CNS-0540147. We acknowledge a faculty grant from NVIDIA Corporation, who supplied the GPU hardware on which this work was performed.

The authors are with Computer Sciences Department, University of Wisconsin, 1210 W. Dayton Street, Madison, WI 53706, USA (e-mail: sklee@cs.wisc.edu; website: www.cs.wisc.edu/~sklee; e-mail: swright@cs.wisc.edu; website: www.cs.wisc.edu/~swright).

In this paper we introduce NVIDIA's GPUs and describe the CUDA software platform that can be used to implement algorithms on these GPUs. We then describe the applications and algorithms, discuss some details of the GPU implementations, and give computational results that compare the speed of implementations on the CPU host with the speed of the GPU implementation. Since the cost of GPUs is so low (hundreds of dollars) and the speedups so high (two orders of magnitude and more), GPU implementations provide a remarkable extension of numerical computing capabilities in certain areas at an extremely low cost.

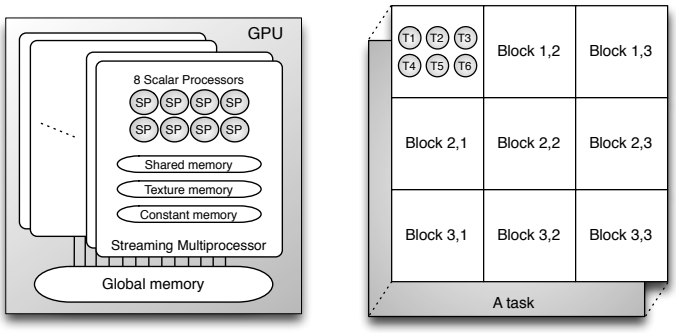
II. Hardware and Software Platform

A. NVIDIA GPUs

NVIDIA provides a wide range of GPU products that can be used for general computing as well as for their original purpose of real-time rendering or 3D graphics. Here we briefly describe two recent models, the GeForce 9800 GX2 and Tesla D870. Both devices provide two GPUs whose processor and memory specifications are summarized in Table I. In both products, each GPU has 128 processors called *scalar processors* (SPs), which run threads and access the shared or global memory concurrently. A *streaming multiprocessor* (SM) is composed of 8 SPs together with an instruction unit, 8192 registers, shared memory, and caches. Each GPU has a high-bandwidth global memory, which can be accessed from all 128 SPs in the same GPU.

A schematic view of a GPU composed of these elements is shown in Fig. 1(a). In these products, the memory bandwidth of the global memory in GPUs is much higher than the bandwidth of the host memory: GPU memory transfer rate is more than 60GB/s, whereas the host memory transfer rate is 6.4GB/s for DDR2-800 memory (the memory on our host PC), or 12.8GB/s for DDR3-1600 memory (the fastest presently available.) The GPUs are connected to a host computer via the PCI Express bus, which provides the maximum bandwidth of 8GB/s (PCI Express v2.0 $\times 16$) between the host and GPU. Although this rate is much lower than the GPU-to-GPU transfer rate, it is actually higher than the host-to-host memory transfer rate for the DDR2-800 host memory that we use. Our host computer is a Dell Precision T5400 workstation, equipped with a 2.66GHz Intel quad-core processor and 4GB of main memory.

Since our focus in this paper is on fine-grained parallelization of the algorithms in question, we use only a single GPU of the GeForce 9800 GX2 device. Multiple GPUs can be utilized if we overlay a coarse-grained parallelization on the



(a) A schematic view of a GPU (b) A thread execution configuration

Fig. 1. The composition of a GPU and a CUDA task grid.

Device name	GeForce 9800 GX2	Tesla D870
# GPUs	2	2
# Streaming Multiprocessors	16 ($\times$2)	16 ($\times$2)
Number of Scalar Processors	8×16 ($\times$2)	8×16 ($\times$2)
Memory size	512MB ($\times$2)	1.5GB ($\times$2)
GPU memory bandwidth	64GB/s ($\times$2)	76.8GB/s ($\times$2)
Host communication bandwidth	8GB/s	4GB/s

TABLE I
THE SPECIFICATIONS OF NVIDIA GPUS.

algorithms described below. This could be accomplished by writing CPU-based parallel codes using pthread, MPI, or other parallel tools, but since such techniques are well understood, we do not investigate them in this paper.

B. Software Platform

The Compute Unified Device Architecture (CUDA) is NVIDIA's software platform for GPUs [9]. It is an extension to the the C++ language that allows users to write thread execution configurations, manage device memory, and do thread synchronization. CUDA splits a task into a grid of *blocks*, where a block is composed of a set of threads. (An example of a grid composed of nine blocks, each with six threads, is shown in Fig. 1(b).) These blocks are scheduled to run on the multiprocessors of the GPU. CUDA uses a single-instruction multiple-thread (SIMT) model, which means that the threads running in a multiprocessor share the same code but run at possibly different states with different streams of data. CUDA also provides GPU-accelerated libraries for basic linear algebra operations (CUBLAS [10]) and discrete Fourier transforms (CUFFT [11]).

In CUDA, each GPU thread receives its own set of dedicated registers, unlike threads that run on CPUs, which typically share registers. This design helps minimize the costs of context switching on GPUs. When registers are shared, their contents must be stored in memory when one thread leaves a processor, and new values for the entering thread must be loaded from memory. Because memory transfers are about one hundred times slower than computations, avoidance of register content copies adds greatly to the benefits of fine-grained parallelism in CUDA. On the other hand, although multiprocessors contain a huge number of registers, the dedication of registers to threads limits the number of threads that can be simultaneously scheduled on a multiprocessor.

The applications most suited to GPU implementations are those that are intensive in computation, for which the ratio of computation to memory accesses is high. Because CUDA provides faster memory accesses to coalesced patterns and spatially local patterns, operations on dense matrices and two-dimensional image data can be implemented efficiently. Since the shared memory in streaming multiprocessors can be accessed as fast as the registers (provided there is no bank conflict), operations that apply the same small data elements to different locations of a large data set can also be optimized. However, global memory size and host communication speed are limiting factors in any implementations, and both should be controlled carefully. Tasks should be large enough to keep all multiprocessors busy so that the latency of memory operations does not affect the overall efficiency too greatly.

We refer to Appendix A for further details on maximizing the efficiency of CUDA programs on GPUs.

III. COMPRESSED SENSING

A. Problem Description

In compressed sensing, we seek to identify a signal that is known to be *sparse*, that is, when represented as a set of n coefficients in some basis representation, only $S \ll n$ of the coefficients are nonzero. We aim to recover the signal from a set of m observations, with $m < n$, where each observation is some linear function of the signal — a linear combination of the coefficients in the basis representation. The observation vector $y \in \mathbb{R}^m$ can thus be expressed in terms of the signal coefficient vector x by $y = Ax$, where the $m \times n$ matrix A is referred to as the *sensing matrix*. Under certain assumptions on A, we can recover the true x by solving the problem

$$\min \|x\|_1 \text{ subject to } Ax = y, \tag{1}$$

which is the solution of the underdetermined linear system $Ax = y$ with smallest ℓ_1-norm. The crucial assumption on A, referred to as the *restricted isometry property (RIP)*, essentially requires that each column submatrix of A for which the number of columns is comparable to S is almost orthonormal. This property ensures that any two signals of sparsity at most S retain their distinctiveness when operated on by A, so that they give rise to significantly different observation vectors. When A has such a property, the formulation (1) yields the true signal even when m is only a modest multiple of the number S of nonzero coefficients in the signal.

When the observation vector y contains error, such as measurement noise, it is inappropriate to enforce the constraint $Ax = y$ exactly. Here, we replace (1) by the following weighted formulation:

$$\min_x \phi(x) := \frac{1}{2}\|Ax - y\|_2^2 + \tau\|x\|_1, \tag{2}$$

for some regularization parameter $\tau > 0$. Note that for $\tau \geq \tau_{\max}$ the solution is $x = 0$, where

$$\tau_{\max} := \|A^T y\|_\infty. \tag{3}$$

Theory concerning the effectiveness of these formulations for finding sparse solutions of $Ax = y$ can be found in

the papers of Donoho [12], [13], Candès and Tao [14], and Candès, Romberg, and Tao [15]. Donoho [16] and Candès [17] give introductions to compressed sensing and discuss the various contexts in which these problems arise.

B. Algorithms

The optimization formulations above are conceptually simple — (1) can be written as a linear program and (2) as a convex quadratic program, with a suitable splitting of the variable x — but their difficulty arises from their high dimensionality (m and n large) and the fact that A is dense in many applications of interest. Many algorithms have been proposed, the vast majority of which do not require the full matrix A (or significant submatrices of A) to be stored or factored explicitly. Rather, they require numerous matrix-vector products involving A and A^T to be performed. Fortunately, there are interesting matrices A that satisfy RIP for which such products can be calculated economically. For example, if A consists of m rows randomly drawn from an n-dimensional discrete cosine transformation (DCT), the products Au and $A^T v$ can be computed in $O(n \log n)$ operations. The same complexity estimate is true if we work with signals in the complex domain (A, x, and y containing complex elements), where A consists of m rows randomly drawn from a discrete Fourier transform (DFT).

As a sample of the vast algorithmic literature, we mention the iterative thresholding / shrinking (IST) approach [18], [19]; extensions of this approach [20], [21] based on the optimal first-order methodology of Nesterov [22], [23]; a variant of IST that uses a "continuation" strategy of successively reducing the parameter τ in (2) [24]; interior-point methods [25], [26], [27]; and gradient projection applied to the bound-constrained quadratic programming formulation of (2) [28].

We focus in this paper on the SpaRSA approach of [29], which can be viewed as an accelerated variant of IST. From the current iterate x^k, SpaRSA obtains the new iterate x^{k+1} by solving the following subproblem:

$$x^{k+1} = \arg\min_z \frac{1}{2}\alpha_k \|z - x^k\|_2^2 + (z - x^k)^T A^T (Ax^k - y) + \tau \|z\|_1 \tag{4a}$$

$$= \arg\min_z \frac{1}{2} \left\| z - \left[x^k - (1/\alpha_k) A^T (Ax^k - y)\right] \right\|_2^2 + \frac{\tau}{\alpha_k} \|z\|_1, \tag{4b}$$

for some $\alpha_k > 0$. The subproblem (4a) is the same as (2) except that the true Hessian $A^T A$ replaced by $\alpha_k I$, and a constant term is omitted. It is separable in the components of z, so it requires $O(n)$ operations to solve (a closed-form solution is easy to derive), in addition to one matrix-vector multiplication each by A and A^T, which are required to form the term $A^T(Ax^k - y)$.

Different variants of SpaRSA are distinguished by different strategies for choosing α_k. A nonmonotone approach (that does not necessarily yield a decrease in the objective (2) at each iteration) uses a choice of α_k inspired by Barzilai and Borwein [30]. Denoting by $s^k := x^k - x^{k-1}$ the difference between the last two iterates and $y^k := A^T A s^k$ the difference between the gradient of the sum-of-squares term $(1/2)\|Ax - b\|_2^2$ at the last two iterates, this variant sets

$$\alpha_k = \frac{(s^k)^T y^k}{(s^k)^T s^k} = \frac{\|As^k\|_2^2}{\|s^k\|_2^2}. \tag{5}$$

(Note that α_k lies in the spectrum of the Hessian $A^T A$, so in this sense, $\alpha_k I$ is a plausible approximation to $A^T A$.) A monotone variant uses the value (5) as an initial guess, then repeatedly increases α_k by a constant factor $\eta > 1$ until x^{k+1} obtained from (4) has a lower function value than x^k.

The performance of SpaRSA (and other IST methods) generally degrades as the regularization parameter τ is reduced and the solutions x become more dense. Much of the practical efficiency can be recovered, however, by the use of a continuation strategy. We start by using SpaRSA to solve (2) for a larger value of τ, then decrease τ in steps toward its target value, using the solution for the previous value of τ as the starting point for each successive τ value. In the results report below, we specify the number of continuation steps C and step from the starting value $\tau_0 = 0.8\tau_{\max}$ to the target value $\tau_C = \tau$ in C steps $\tau_1, \tau_2, \ldots, \tau_C$, where

$$\log \tau_l = \log \tau_0 + (l/C)^a (\log \tau_C - \log \tau_0). \tag{6}$$

Here, $a \geq 1$ is a parameter governing the "bunching" of τ_l values near the target τ. (The value $a = 1$ leads to constant ratios τ_{l+1}/τ_l, while larger values of a causes these ratios to become larger as l increases.)

Implementations of SpaRSA and other algorithms often include an optional postprocessing step known as *debiasing*, in which the regularization term is dropped from the objective in (2) and an unconstrained least squares problem is solved, with the zero components of x from the main SpaRSA algorithm discarded. The last step is often performed with a conjugate gradient approach, for which the major operations at each iteration are multiplications by A and A^T and some Level 1 BLAS calculations, just as in the main SpaRSA algorithm. We would thus expect the debiasing step to be implemented as efficiently on a GPU as the main SpaRSA algorithm. For simplicity, however, we did not include the debiasing feature in the implementations described in this paper.

C. GPU Implementation

For efficient GPU implementation of SpaRSA, it is important that the sensing matrix A is one that can be stored compactly (and implicitly) and that matrix-vector products involving A and A^T can be formed efficiently on the GPU architecture. The other major operations in SpaRSA — vector additions and inner products — are simple and can be implemented efficiently or using BLAS operations from the CUBLAS library [10]. We use CUBLAS library to compute ℓ_1-norms and inner products, but for all the other BLAS and $O(n)$ operations we use our own codes, as they are marginally more efficient than the corresponding CUBLAS routines. (This is also true for image reconstruction algorithms we discuss later.) The total amount of storage used is a small multiple of n. Specifically, we need to store the current iterate x^k, the candidate for next iterate x^{k+1}, the step s^k, the gradient of the

sum-of-squares term $A^T(Ax^k - y)$ (each of which requires n locations), and the vectors As^k and $Ax^k - y$ (each of which requires m locations), together with whatever storage is needed to represent A.

Compressed sensing experiments often make use of matrices A whose elements consist of numbers drawn independently from a random distribution about 0 with a common variance, or from a Bernoulli distribution (entries ± 1 with equal probability). Such matrices are known to have good restricted isometry properties and thus allow effective recovery of sparse signals. However, they are not practical for GPU implementation, as they generally either require much more than $O(n)$ storage, or are expensive to regenerate whenever needed. Matrices A consisting of randomly chosen rows from a discrete cosine transformation (DCT) (in the case of real data) or a discrete Fourier transform (for complex data) are much more suitable. Besides having satisfactory restricted isometry properties, such matrices can be stored compactly (just m locations are needed, to store the indices of the chosen rows), and implemented using the CUFFT library [11]. DCTs can be implemented using DFTs with $O(n)$ pre- and post-processing steps, which can be executed efficiently on GPUs.

In our GPU implementation, significant data transfer between the host machine and GPUs occurs only at the beginning and the end of the algorithm. The initial values of the variable x, together with the observation vector y, the m row indices defining A, the regularization parameter τ, and possibly some algorithmic parameters are copied to the GPU at the start, and the solution is returned at the end. The host CPU is still used for small computations, such as computation of the roster of τ values used in the continuation process, in scalar comparisons, and in checking of loop control variables. Further savings could be made in data transfers if the value $x = 0$ as the starting point in the continuation process (it usually suffices.) We can also return a compressed version of the solution of (2), consisting of the nonzero components of x and their indices. Further, if numerous instances of (2) are to be solved in sequence for different data, data transfers for successive instances could be overlapped with computation.

D. Computational Results

We discuss results obtained with simple implementations of SpaRSA, applied to problems in which the signal is a sparse one- or two-dimension array, and the sensing matrix consists of randomly selected rows from a discrete cosine transformation (DCT). Noise may be added to the observation vector. Our main point of comparison is between

- the runtime for the Matlab implementation running on the CPU of the host machine (which we refer to hereafter as the "CPU implementation"), and
- the runtime for the CUDA-based GPU implementation.

Both implementations use the same Matlab code to set up the problem and analyze the results, but the GPU code replaces the Matlab routine implementing SpaRSA with a functionally equivalent CUDA routine.

We note several distinctions between the two implementations. First, the GPU computations are carried out in single precision, while the CPU computations are in double precision. This is because the GPU cards we used support only single-precision arithmetic. (Double-precision cards are only now coming on the market, and the CUDA environment and its libraries are being extended to run on them.) The lower accuracy of single precision computations creates significant issues for the GPU implementation of SpaRSA. The duality-based stopping criterion for each value of τ described in [29] cannot be implemented satisfactorily in CUDA. Inaccuracy in computation of the gap often makes it difficult to reduce the relative gap below about 10^{-4}, regardless of how many iterations are performed, for smaller values of τ; no such difficulties are observed in the double-precision CPU implementation. In our implementation, we used a stopping criterion for each value of τ based on the relative change in objective value from one iteration to the next, namely,

$$\frac{|\phi(x^k) - \phi(x^{k-1})|}{\phi(x^k)} \leq 10^{-6}. \tag{7}$$

Still, as we see in the tables below, the GPU implementation sometimes requires slightly more iterations than the CPU implementation to find a solution of equivalent accuracy.

A second difference between the implementations is the use of Matlab code for the CPU implementation vs. CUDA code (an extension of C++) for the GPU implementation. We believe however that a CPU implementation via C++ and mex files would show little if any improvement over the Matlab implementation. The major computational operations are DFT, DCT, and Level 1 BLAS operations, all of which are implemented with high efficiency in Matlab. In fact, we could speed up the GPU implementation further by using C++ calling code in place of the Matlab and mex software. Multiple data transfer requests on page-locked host memory could be made without having to wait for the completion of the previous requests, making room for other jobs in the host or GPUs. See Appendix A for further details.

We note several issues regarding initialization of GPU computations. The very first call to a CUDA utility function made by a process (for example, `cudaGetDeviceCount()` which counts the number of CUDA-enabled GPUs, or `cudaSetDevice()` which selects a particular GPU) executes in about .01 seconds. The very first GPU memory allocation call issued by a process takes significantly longer than subsequence memory allocation calls: about .8 seconds. Following these initializations, GPUs still appear to need some "warming up" before they reach peak performance. For example, if we compute the ℓ_1-norm of a vector of length 10^6, the first call takes approximately 2.5 times as long as the second call. These overheads will be amortized over all the invocations of SpaRSA made by that process. In real situations, we can expect SpaRSA to be invoked multiple times to process different data sets (for example, different time slices of a signal) in sequence, or possibly to be used as one of a number of tasks running concurrently on the GPUs. Hence, we do not include these initialization overheads in our discussion of GPU execution times below.

1) One-Dimensional Signal with DCT Sensing: Our first experiment is with a one-dimensional signal with n compo-

$\tau/\tau_{\max}$	CPU			GPU			Speedup	
	iters	time (s)	MSE	iters	time (s)	MSE	total	iter
.000100	103	4.32	8.1e-10	129	0.16	7.2e-10	26	33
.000033	135	5.52	1.3e-10	126	0.15	2.0e-10	37	34
.000010	143	5.81	9.8e-11	139	0.17	1.3e-10	35	34

TABLE II
COMPUTATIONAL RESULTS FOR A 1-D DCT SENSING MATRIX OF DIMENSION 8192×65536, WITH 1638 SPIKES

$\tau/\tau_{\max}$	CPU			GPU			Speedup	
	iters	time (s)	MSE	iters	time (s)	MSE	total	iter
0.000100	107	107.08	9.1e-10	129	2.08	8.5e-10	51	62
0.000033	131	129.10	1.7e-10	131	2.10	1.6e-10	61	61
0.000010	149	145.31	1.0e-10	160	2.57	9.0e-11	57	61

TABLE III
COMPUTATIONAL RESULTS FOR A 1-D DCT SENSING MATRIX OF DIMENSION 131072×1048576, WITH 26214 SPIKES

$\tau/\tau_{\max}$	CPU			GPU			Speedup	
	iters	time (s)	MSE	iters	time (s)	MSE	total	iter
0.10	62	2.56	1.9e-05	67	0.09	1.9e-05	27	30
0.05	67	2.68	4.8e-06	68	0.08	4.8e-06	32	32
0.02	77	3.06	9.7e-07	83	0.10	9.6e-07	30	32

TABLE IV
COMPUTATIONAL RESULTS FOR A 2-D DCT SENSING MATRIX OF DIMENSION 1311×65536, WITH 60 SPIKES

$\tau/\tau_{\max}$	CPU			GPU			Speedup	
	iters	time (s)	MSE	iters	time (s)	MSE	total	iter
0.10	64	98.25	3.2e-05	64	1.00	3.1e-05	98	98
0.05	65	103.10	7.9e-06	70	1.08	7.9e-06	95	102
0.02	80	117.97	1.5e-06	84	1.30	1.5e-06	91	95

TABLE V
COMPUTATIONAL RESULTS FOR A 2-D DCT SENSING MATRIX OF DIMENSION 20972×1048576, WITH 1031 SPIKES

nents, in which the sensing matrix consists of $m < n$ rows drawn randomly from an $n \times n$ DCT matrix. The signal consists of $\lfloor m/5 \rfloor$ spikes, half of which have magnitude near 1 with the remaining half having magnitudes between 10^{-5} and 10^{-4} (logarithms uniformly distributed). There is also noise in the signal of order 10^{-6} on each element.

Results are shown in Tables II and III. In both tables, the SpaRSA algorithm was run with 10 steps of continuation, with three different target values of τ, shown in the first column of each table. The total iteration counts and for the GPU and CPU-based implementations are shown, as is the final mean-square error in the recovered solution. The quality of the solution is similar for both GPU and CPU implementations. Two speedup figures are shown in the final columns of each table. The first is calculated by comparing total runtimes between the GPU and CPU implementations, and the second by comparing runtimes per iteration. Runtimes include memory transfer time.

Table II shows results for a sensing matrix of dimension $2^{13} \times 2^{16}$. For each value of τ, the GPU execution time was approximately 0.16 seconds, whereas the CPU implementations required about 5 seconds, resulting in speedups of 26 to 37 in total time. The average speedups per iteration are about 34, and are almost identical for all τ values, indicating that the speedup resulting from use of the GPU is quite stable and consistent. Note that each iteration of SpaRSA on the GPU (in which the main computational effort is two matrix-vector operations involving the sensing matrix) takes about 1.2 microseconds, versus about 40 microseconds for the CPU implementation.

Table III shows results for a sensing matrix of dimension $2^{17} \times 2^{20}$, a factor of 16 larger in each dimension. Here the speedups are 51 to 61 (counting total runtime), and about 61 on average per iteration. The memory transfer between host and GPU takes more time, but this cost is amortized over a much longer execution time, so its effects on speedup figures is less deleterious. Moreover, the larger data set yields a larger number of thread blocks in our implementation, resulting in less idle time on the GPU's multiprocessors. Each iteration of SpaRSA in the GPU implementation requires about 16 microseconds, compared to about one second in the CPU implementation.

2) Two-Dimensional Signal with DCT Sensing: Our second result is for a sparse two-dimensional signal of length n, which can be thought of as an $\bar{n} \times \bar{n}$ array of pixels where $n = \bar{n}^2$, only a small fraction of which are nonzero spikes. The sensing matrix consists of m rows randomly selected from an two-dimensional ($\bar{n} \times \bar{n}$) DCT matrix. We choose the fraction of spikes to be .001, and set m to be 20 times the number of spikes. Each spike is chosen randomly to be $+1$ or -1, and the observations are corrupted by noise drawn independently from a Normal distribution, $\mathcal{N}(0, .001^2)$.

The values of regularization parameter τ on which we report are approximately the ones needed to produce results of good quality. For $\tau = .05\tau_{\max}$ and $\tau = .1\tau_{\max}$ where $\tau_{\max}$ is defined in (3), the algorithms recover approximately the same number of nonzero components as in the true solution. For $\tau = .02\tau_{\max}$ we have a slightly under-regularized solution, with more spikes than the true solution.

We report results in Table IV for a problem with $n = 2^{16}$ and $m = 1311$, while Table V shows a larger problem with $n = 2^{20}$ and $m = 20972$. The relative performances of the two approaches are broadly similar to the one-dimensional case. For larger data sets, the advantage of the GPU implementation increases, as we discussed before.

IV. IMAGE RESTORATION

Image restoration is an important task in image segmentation or computer vision applications. Regularization methodologies based on total variation (TV), introduced by Rudin, Osher, and Fatemi [31], are highly effective in removing noise, blur, or other unwanted fine-scale detail, while preserving edges. We report here on TV-regularized denoising and deblurring formulations, solved with a particularly effective primal-dual gradient descent approach described recently by Zhu and Chan [32]. The main computational operations in this algorithm include a difference operation (needed to calculate the TV-norm), DFTs and inverse DFTs (needed in the deblurring application), and various Level 1 BLAS operations and simple projection operations. All these operations require $O(N)$ operations, where N is the number of unknowns, with the exception of the DFT and inverse DFT procedures, which require $O(N \log N)$ operations. All can be implemented efficiently on a GPU.

After briefly discussing the formulation and the algorithms, we compare the efficiencies of CPU and GPU implementations.

A. Denoising

Given a domain $\Omega \subset \mathbb{R}^2$ (usually a rectangle) and an observed image $f : \Omega \to \mathbb{R}$, we recover a denoised image $u : \Omega \to \mathbb{R}$ by solving the following problem in the appropriate function space:

$$\min_u \int_\Omega |\nabla u|_2 + \frac{\bar{\lambda}}{2}\|u - f\|_2^2, \tag{8}$$

where $\bar{\lambda}$ is a regularization parameter. (Larger values of $\bar{\lambda}$ yield better fidelity to the recorded image, while smaller values produce more cartoon-like images, with larger areas of constant intensity.) The notation $|\cdot|_2$ represents the Euclidean norm on $\mathbb{R}^2$, and the first term in the expression (8) is the TV seminorm of u.

We can obtain saddle-point and dual formulations of (8) in function space by an appropriate redefinition of the TV seminorm. (Some details are given in [33], for example.) We focus here on a simple finite-difference discretization of this problem and its corresponding saddle-point (min-max) and dual formulations.

Assume for simplicity that Ω is square, and define an $n \times n$ grid of pixels over the domain, indexed by (i, j), where $i, j = 1, 2, \ldots, n$. The unknown function u is replaced by an $n \times n$ matrix u_{ij}, and the discrete spatial gradient ∇u is defined by

$$(\nabla u)^1_{i,j} = \begin{cases} u_{i+1,j} - u_{i,j} & \text{if } i < n \\ 0 & \text{if } i = n \end{cases} \tag{9a}$$

$$(\nabla u)^2_{i,j} = \begin{cases} u_{i,j+1} - u_{i,j} & \text{if } j < n \\ 0 & \text{if } j = n. \end{cases} \tag{9b}$$

The discretized TV seminorm is thus

$$\mathrm{TV}(u) = \sum_{1 \le i,j \le n} \|(\nabla u)_{i,j}\|_2. \tag{10}$$

We can obtain a discrete version of the formulation (8) by reshaping the unknown matrix u into a vector $v \in \mathbb{R}^N$ (where $N = n^2$), defined as follows:

$$v_{(j-1)n+i} = u_{i,j}, \quad 1 \le i, j \le n.$$

The (i, j) component of the gradient (9) can thus be represented as a multiplication of the vector $v \in \mathbb{R}^N$ by a matrix $A_l^T \in \mathbb{R}^{2\times N}$, for $l = 1, 2, \ldots, N$:

$$A_l^T v = \begin{cases} (v_{l+1} - v_l;\ v_{l+n} - v_l) & \text{if } l \bmod n \neq 0 \text{ and } l + n \le N \\ (0;\ v_{l+n} - v_l) & \text{if } l \bmod n = 0 \text{ and } l + n \le N \\ (v_{l+1} - v_l;\ 0) & \text{if } l \bmod n \neq 0 \text{ and } l + n > N \\ (0;\ 0) & \text{if } l \bmod n = 0 \text{ and } l + n > N. \end{cases} \tag{11}$$

Using this notation, the discretization of (8) can be written as follows:

$$\min_v P(v) := \sum_{l=1}^N \|A_l^T v\|_2 + \frac{\lambda}{2}\|v - g\|_2^2, \tag{12}$$

where g represents a discretization of the image f and λ is an appropriately scaled version of $\bar{\lambda}$. We obtain the min-max form by introducing vectors $x_l \in \mathbb{R}^2$, $l = 1, 2, \ldots, N$ and noting that

$$\|A_l^T v\|_2 = \max_{\|x_l\|_2 \le 1} x_l^T A_l^T v.$$

By defining

$$x := (x_1; x_2; \ldots; x_N) \in \mathbb{R}^{2N},$$
$$A := [A_1 \,|\, A_2 \,|\, \ldots \,|\, A_N],$$
$$X := \{x \in \mathbb{R}^{2N} \,|\, \|x_l\|_2 \le 1,\ l = 1, 2, \ldots, N\},$$

we can write the min-max formulation of (12) as follows:

$$\min_v \max_{x \in X} \ell(v, x) := x^T A^T v + \frac{\lambda}{2}\|v - g\|_2^2. \tag{13}$$

Interchanging min and max, and solving the minimization explicitly for v, yields the following dual formulation:

$$\max_{x \in X} D(x) := \left[\frac{\lambda}{2}\|g\|_2^2 - \frac{1}{2\lambda}\|Ax - \lambda g\|_2^2\right], \tag{14}$$

which we can write equivalently as

$$\min_{x \in X} \frac{1}{2}\|Ax - \lambda g\|_2^2. \tag{15}$$

Algorithms for solving the dual formulations (14) and (15) — mainly algorithms of gradient projection type, with different choices of steplength — are described in [33]. These first-order methods are shown to be effective for finding solutions of low to moderate accuracy, though they tend to be overtaken by second-order methods such as the one described in [34] when high accuracy is demanded. The first order methods require at each calculation of the gradient residual $r := Ax - \lambda g$ and the gradient of (15), which is $A^T r$. Multiplication by A^T is the difference operation essentially defined by (9); multiplication by A is a discretized divergence operation, defined correspondingly. In neither case is any explicit storage required for A, and both operations can be performed with high efficiency on a GPU, as we discuss below. The other major operations required for gradient projection methods are projections onto X (which are $O(N)$ operations and are also easy to perform on GPUs) and Level-1 BLAS operations.

We focus here on a primal-dual hybrid (PDHG) gradient projection approach proposed by Zhu and Chan [32], which requires the same basic operations as the dual gradient projection approaches. This method has striking performance on practical denoising problems, outstripping other first-order methods, and even second-order methods, regardless of the solution accuracy demanded. The method generates a primal-dual sequence $(v^k, x^k) \in \mathbb{R}^N \times X$ by taking the following steps:

$$x^{k+1} := P_X(x^k + \tau_k \nabla_x \ell(v^k, x^k)) \tag{16a}$$

$$v^{k+1} := v^k - \sigma_k \nabla_v \ell(v^k, x^{k+1}), \tag{16b}$$

where $P_X(\cdot)$ denotes projection onto the set X and τ_k and σ_k are positive steplengths. These are gradient ascent/descent steps taken alternately in dual and primal variables, projected

onto the appropriate feasible set. The results reported in [32] are obtained with the following steplengths:

$$\tau_k := (.2 + .08k)\lambda, \qquad \sigma_k := \frac{1}{\tau_k}\left(.5 - \frac{1}{3 + .2k}\right). \quad (17)$$

Note in particular that, somewhat counterintuitively, we have $\tau_k \to \infty$. However, the projection onto X in (16a) keeps the steps between successive iterates x^k short on later iterations. The theoretical properties of this approach are not well understood, though some connections with literature on variational inequalities and saddle-point problems are pointed out in [32].

We declare numerical convergence of the algorithm when the relative duality gap falls below a specified level, specifically:

$$\frac{P(v) - D(x)}{|P(v)| + |D(x)|} \leq \text{TOL}. \quad (18)$$

B. Deblurring

When the problem data f is not the image itself but some blurred version of it involving a known linear blur operator $\mathcal{K}$, the problem (8) becomes

$$\min_u P(u) := \int_\Omega |\nabla u|_2 + \frac{\bar{\lambda}}{2}\|\mathcal{K}u - f\|_2^2. \quad (19)$$

By using the same discretization of the regularization term as in (9) and (10), and replacing $\mathcal{K}$ by a discretization K, we obtain the following discrete form of (19):

$$\min_v P(v) := \sum_{l=1}^{N} \|A_l^T v\|_2 + \frac{\lambda}{2}\|Kv - g\|_2^2, \quad (20)$$

where λ is an appropriately scaled version of $\bar{\lambda}$. The min-max form, generalizing (13), is as follows:

$$\min_v \max_{x \in X} \ell(v, x) := x^T A^T v + \frac{\lambda}{2}\|Kv - g\|_2^2. \quad (21)$$

Zhu and Chan [32] note that the blurring operator $\mathcal{K}$ is ill-posed in many applications, and that the PDHG steps (16) do not give rapid convergence. Good performance can be recovered, however, by making the step in the variable v semi-implicit, modifying (16) as follows:

$$x^{k+1} := P_X(x^k + \tau_k \nabla_x \ell(v^k, x^k)) \quad (22a)$$

$$v^{k+1} := v^k - \sigma_k \nabla_v \ell(v^{k+1}, x^{k+1}). \quad (22b)$$

Note that v^{k+1} appears on the right-hand side of (22b). This formula can be implemented efficiently in terms of forward and inverse DFTs as follows:

$$v^{k+1} = \mathcal{F}^{-1}\left[\frac{\mathcal{F}(v^k - \sigma_k A x^{k+1}) + \sigma_k \lambda \overline{\mathcal{F}(K)}\mathcal{F}(g)}{1 + \sigma_k \lambda \overline{\mathcal{F}(K)}\mathcal{F}(K)}\right], \quad (23)$$

where $\mathcal{F}(\cdot)$ and $\mathcal{F}^{-1}(\cdot)$ represent the forward and inverse DFT operators respectively, and $\overline{B}$ denotes the complex conjugate of a matrix B. All matrix-matrix multiplications and divisions are pointwise and so can be performed in $O(N)$ operations. The following choice of parameters was used in [32], and we use them also in our implementations:

$$\tau_k := 10 + 40k, \qquad \sigma_k := \frac{1}{\tau_k}\left(1 - \frac{.2}{k}\right). \quad (24)$$

C. GPU Implementation

In both the denoising and deblurring algorithms, the spatial difference operation (11) and its transpose, the discrete divergence operation, are among the most time-dominant operations. Our GPU implementation of these operations is divided into N threads, each of which produces entries for a single value of the index ℓ. Considering the operations in two dimensions, the thread that produces the (i, j)-th output entry has to access not only the (i, j) location of the input vector, but also adjacent locations of the input vector in the i and j directions, which are also needed by other threads. Rather than having each thread doing its own memory fetches — which would result in the same location of the input vector being fetched multiple times — we impose a cache memory called a *texture* on global memory to reuse fetched entries, which greatly reduces the average memory latency. Texture memory is illustrated in Fig. 1(a), while more details on the use of textures to reduce average memory latency are given in Appendix A.

At various places in the algorithms, we need to perform a reduction of a data set of size proportional to the problem dimension to a single value. For example, to compute the primal and the dual objective function values in the denoising algorithm, we have to sum n^2 values. Careful implementation is required to perform such operations efficiently on a GPU. We can divide the computation into thread blocks, where each block performs a partial reduction and produces a single number as a result. To make the most efficient use of GPU multiprocessors, we should use enough thread blocks to keep 16 multiprocessors busy. There is a tradeoff, however. Since each thread block must store its result in a global memory location, we would pay a high price in memory latency if too many blocks were used, and space in global memory is limited in any case.

In our implementation of a global sum operation, each thread block of 256 threads first reads 256 global memory entires in coalesced fashion and stores them in a shared memory block of dimension 16×16. The block performs the same operation seven more times, adding each 16×16 block fetched from global memory to the 16×16 block resident in shared memory. The thread block then sums up the entries in this 16×16 block, producing a single number — the sum of 2048 numbers in all — which it stores in global memory. In this fashion, we produce a global memory vector containing $n^2/2048$ partial sums. The whole procedure is then repeated recursively, reducing the dimension of the stored vector by a factor of 2048 each time, until we obtain the result. For example, we can sum a vector of dimension up to 2^{22} by performing two stages of this partial summation procedure. The global memory space needed to store the vector of partial sums after the first pass of reduction would consist of 2^{11} locations, or about 8KB in single precision.

Elementary operations that are not dependent on each other can be executed concurrently in a GPU by means of *streams*. A stream is the unit of streamlined synchronization in CUDA; a synchronization point can be defined for multiple streams by a call to the CUDA function `cudaThreadSynchronize()`.

Image size	Tol	CPU		GPU		Speedup	
		iters	time (s)	iters	time (s)	total	iter
128^2	1.e-2	11	0.03	11	0.02	2	2
	1.e-4	79	0.21	79	0.02	11	11
	1.e-6	338	0.90	329	0.07	14	13
256^2	1.e-2	13	0.17	13	0.02	9	9
	1.e-4	68	0.81	68	0.03	32	32
	1.e-6	304	3.57	347	0.11	33	38
512^2	1.e-2	12	0.95	12	0.03	31	31
	1.e-4	54	3.96	54	0.05	76	76
	1.e-6	222	16.08	238	0.19	84	90
1024^2	1.e-2	14	5.42	14	0.08	64	64
	1.e-4	69	25.80	69	0.24	106	106
	1.e-6	296	103.54	324	1.02	102	111
2048^2	1.e-2	13	31.41	13	0.28	114	114
	1.e-4	67	149.24	67	0.90	165	165
	1.e-6	319	694.16	338	4.12	169	179

TABLE VI
COMPUTATIONAL RESULTS OF IMAGE DENOISING (λ=0.041.)

Using streams, we can for example update the rows of the dual variable x simultaneously, since the updates for the different rows are independent.

For the deblurring algorithm, we prepare the blur kernels K in Matlab and pass them to both CPU and GPU codes. These kernels are created by using the Matlab function `fspecial()`, followed by a call to `fftshift()`, to shift the zero frequency components to the center of spectrum by calling. The kernels are padded with zero values by `padarray()` Matlab function, so that the size of the kernel data structure matches that of the image data structures. (All three routines are available in the Matlab image processing toolbox.) In the GPU implementation, we incur some inefficiency by transferring the padded kernel to the GPU, as it contains mostly zeros, but the effect on overall runtime is minor.

In our GPU implementations, significant data transfer between the host machine and GPUs occur only at the beginning and at the end of the algorithm as before. We transfer the initial value of the variables to the GPU at the start of the call and then transfer final values at the end. (We can avoid the first transfer by using fixed initial values such as $x = 0$, but the savings are minimal.) For deblurring, we also precompute the values of $\overline{\mathcal{F}(K)}\mathcal{F}(g)$ and $\overline{\mathcal{F}(K)}\mathcal{F}(K)$ at the start of both the CPU and GPU codes, and use them repeatedly in performing the operation (23). Additional storage is needed for these vectors (the GPU implementation stores them in the GPU global memory), but the space is available for problems of the dimensions we consider, and considerable computational savings are made by precomputing these vectors. In the deblurring problem, we use CUFFT library [11] to compute two-dimensional DFTs.

D. Computational Results

For both problems, we used five test images of different sizes: Shape (128×128), Cameraman (256×256), Barbara (512×512), Man (1024×1024), and Earth (2048×2048). These data sets are available from a URL listed in Section V.

1) Denoising: We prepared noisy versions of the test images by adding Gaussian noise of mean 0 and standard deviation 0.1 to the images. The value of λ was fixed to 0.041 for all case. In some cases, better visual results could be obtained with slightly different values of λ, but our focus in this paper is on the relative performance of CPU and GPU implementations, rather than on the efficacy of the formulations, and our conclusions about relative efficiency are not much affected by the choice of λ.

Image size	Blur kernel	CPU		GPU		Speedup	
		iters	time (s)	iters	time (s)	total	iter
128^2	m-motion	31	0.15	31	0.02	6	6
	s-motion	106	0.49	106	0.05	10	10
	m-Gaussian	88	0.41	88	0.04	10	10
	s-Gaussian	66	0.32	66	0.04	9	9
256^2	m-motion	27	0.55	27	0.04	14	14
	s-motion	79	1.57	79	0.08	20	20
	m-Gaussian	44	0.88	44	0.05	17	17
	s-Gaussian	39	0.79	39	0.05	17	17
512^2	m-motion	34	3.94	34	0.14	28	28
	s-motion	72	8.23	72	0.26	31	31
	m-Gaussian	44	5.07	44	0.17	29	29
	s-Gaussian	37	4.27	37	0.15	29	29
1024^2	m-motion	31	19.39	30	0.42	46	45
	s-motion	75	46.00	74	0.95	48	48
	m-Gaussian	44	27.07	44	0.59	46	46
	s-Gaussian	41	24.76	41	0.55	45	45
2048^2	m-motion	33	113.38	33	2.31	49	49
	s-motion	79	263.07	79	5.26	50	50
	m-Gaussian	49	166.36	49	3.34	50	50
	s-Gaussian	48	163.72	48	3.28	50	50

TABLE VII
COMPUTATIONAL RESULTS OF IMAGE DEBLURRING. 'M-' DENOTES 'MILD' AND 'S-' DENOTES 'SEVERE', WHILE 'MOTION' AND 'GAUSSIAN' INDICATE THE TYPE OF BLUR KERNEL.

Here we adopted a stopping criterion based on the duality gap, since no significant precision issues were found in computation for the problems and the parameter values we tried. Several different values of the duality gap tolerance TOL in (18) were tried: 10^{-2}, 10^{-4} and 10^{-6}. The CPU algorithm uses double-precision, whereas the GPU version uses single-precision. The difference in precision resulted in some variation in the number of iterations required to reach the specified accuracy, but the GPU implementation rarely required more than 10% more iterations than the CPU implementation. The speedup varies from 14 to 169 at highest precision, with higher speedups as the image dimension grows. The absolute speed is remarkable; even for the largest image (dimension 2048×2048), the GPU implementation requires less than one second to denoise the image to a moderate tolerance 10^{-4}.

2) Deblurring: We prepared blurred noisy images by first convolving the images with blur kernels, and then by adding Gaussian noise with mean 0 and standard deviation $\sigma = 0.001$. Two types of blur kernels were generated by the Matlab function `fspecial()`: motion blur and Gaussian blur. For each kernel, two settings were used. For motion blur, we used mild (length=21) and severe (length=91) blur, both with an angle of 135 degrees. For the Gaussian blur, mild (size=21, sigma=5) and severe (size=41, sigma=10) settings were used. For all cases, the values of λ were chosen to be $\min\{.2/\sigma^2, 2\times10^{11}\}$. Convergence was declared when the difference between the last two primal iterates, measured by $||v^{k+1}-v^k||_\infty$, fell below 10^{-3}.

Runtime comparisons of the host-based and the GPU-based algorithms are shown in Table VII. Note that for CPU-based implementation, we use `fft2()` function in Matlab, of which

the performance is optimized in a special way when the data set size is a large power of 2, between 2^{14} and 2^{22}. All our images are able to take advantage of this special optimization in the Matlab CPU implementation. Still, our GPU implementation runs between 6 and 50 times faster than the CPU Matlab code. Note also that the runtime of GPU-based code increases almost linearly as the image size quadruples, indicating good scaling of our GPU-based implementation.

V. Conclusions

We have shown that two signal reconstruction problems of current interest in computational science — compressed sensing and image processing — can be solved with high efficiency on commodity GPUs attached to PC platforms. In each case we worked with algorithms that are among the most effective available, even in their CPU implementations. There was no need to revert to less efficient but more parallelizable alternative strategies.

It can be expected that other problems and algorithms with similar characteristics — compute-intensive algorithms, locality of data access, total data size comparable to the number of variables in the problem — can be solved on GPU platforms with similar efficiency.

The codes and data sets used to perform the experiments described in Sections III and IV are available from the URL http://www.cs.wisc.edu/~swright/GPUreconstruction.

Appendix

A. Performance hints for CUDA implementations

We summarize here a number of points that are important to understand in taking full advantage of the potential of GPU implementations. More details can be found in the CUDA documentation [9, Chapter 5].

a) Memory coalescing: Global memory access throughput can be maximized by exploiting memory coalescing. Threads in CUDA are aligned by units called *warps*, which consists of 32 threads for the devices in Table I. If 16 concurrent memory accesses, each of them from a thread of a *half-warp*, are aligned and contiguous, then those 16 memory accesses are combined into a single memory operation automatically by CUDA — an operation that would take 16 times as long if the accesses are performed serially. Serialization of memory accesses should be avoided whenever possible, since global memory access latency is very high. For example, to read a single-precision number from global memory, we have to spend 4 GPU clock cycles to issue the read command and another 400 to 600 cycles to actually fetch the value. By comparison, computations and shared memory accesses in GPUs are much faster. For instance, writing a single-precision number to shared memory takes 4 clock cycles, while a floating-point multiplication takes 4 cycles. It is essential, therefore, to take advantage of memory coalescing in order to maximize performance.

b) Avoid bank conflicts: The shared memory in a streaming multiprocessor can be accessed as fast as registers provided there are no *bank conflicts*. The shared memory on a streaming multiprocessor consists of sixteen equally-size banks, which can be accessed simultaneously. That is, if sixteen threads running on the multiprocessor access sixteen different shared memory locations that fall into different banks, those accesses are performed concurrently. In our experience, bank conflicts do not degrade performance as much as uncoalesced global memory accesses because shared memory is on-chip and much faster than the global memory. Still, it can be an important factor, and we were careful to avoid bank conflicts in our implementations.

c) Use textures for spatially local memory accesses: Global memory is not cached on GPUs. Inefficiencies can result from multiple fetches of the same memory location, when this location is used by several different threads. Such nearly concurrent accesses happen in our image processing solvers of Section IV, where the spatial difference operator (11) and the divergence operator need to access adjacent locations in the two-dimensional vector of unknowns. We could use shared memory to cache these global memory accesses, but CUDA provides much easier-to-use caching facilities by means of the texture memory illustrated in Fig. 1(a). A texture provides a read-only cache for global memory. A reference to a location in this cache has to be associated with a global memory pointer; afterwards global memory can be fetched using caches via this reference.

d) Increase GPU utilization: If only one thread block is scheduled to run on a multiprocessor, the multiprocessor will be idle while threads in the block are waiting for synchronization or for completion of memory accesses. To avoid this idle time, which degrades efficiency, two or more blocks per multiprocessor should be active at any given time. In fact, it is recommended in [9] to have at least one hundred blocks per task to ensure overlapped execution of threads.

A convenient way to monitor the factors described above is to use the CUDA Visual Profiler [35], which shows the numbers of uncoalesced global memory loads/stores, GPU utilization, and other important information.

e) Use page-locked memory: Host memory is pageable in typical operating systems for personal computers, so that the physical memory assigned to a process can be reclaimed at any time by the operating system. CUDA allows host memory to be *page-locked*, enabling data transfers between host and GPU memory to be sped up by a factor of about two. We do not use this feature, however, since our data are stored in the host memory that is managed by Matlab, so is unavailable to CUDA unless we perform costly host memory copies.

B. Speedup of elementary operations

It is often the case that a small fraction of the operations in an algorithm are responsible for the majority of the total runtime. In our CPU implementation of 2-D compressed sensing, the DCT and the inverse DCT operations together take about 96% of the total runtime. Overall speedup depends largely on the speedup of these time-dominant operations.

In Table VIII, we show the execution time of some selected operations in a 2-D compressed sensing run, for the problem with a sensing matrix of dimension 20972×1048576 with 1031 spikes, on CPU and GPU. The "fraction" column in

Operations	CPU		GPU		Speedup
	time (s)	fraction	time (s)	fraction	
2-D DCT	31.47	0.31	0.48	0.44	71
2-D inv. DCT	66.39	0.65	0.54	0.49	133
inner product	0.18	0.0017	0.016	0.015	14
ℓ_1-norm	2.32	0.023	0.010	0.0093	356
ℓ_∞-norm	0.0042	0.000040	0.00014	0.00012	32
Total time	103.36	1.0	1.095	1.0	102

TABLE VIII
GPU ACCELERATION OF OPERATIONS IN 2-D COMPRESSED SENSING.

Operations	CPU time (s)	GPU time	Speedup
2-D DFT	5.01	0.16	31
2-D inv. DFT	5.43	0.16	34
Fraction	.43	.58	-

TABLE IX
GPU ACCELERATION OF OPERATIONS IN DEBLURRING.

these tables shows the fraction of total execution time for the SpaRSA implementation that was consumed by each operation. The speedups for DCT and inverse DCT on a vector of dimension $2^{10} \times 2^{10}$ were 71 and 133, respectively, over the `dct2()` and `idct2()` functions in Matlab. These speedups were the main contributors to the overall speedups of up to 102 seen in Table V for problems of this size. Speedups are less dramatic on some less significant operations, such as inner product computations and ℓ_∞-norm calculations. The ℓ_1-norm calculation in Matlab is inefficient, as can be seen by the speedup of 356 attained by the GPU implementation of this operation. A comparison of a C++ implementation of this operation on the host CPU with the CUBLAS GPU implementation shows a speedup of only 18.

The DFTs using CUFFT library [11] are about 30 times faster than the `fft2()` and `ifft2()` functions in Matlab when applied to a 2-D signal of size 1024×1024, as shown in Table IX.

REFERENCES

[1] B. Catanzaro, N. Sundaram, and K. C. Keutzer, "Fast support vector machine training and classification on graphics processors," in *International Conference on Machine Learning*, 2008.

[2] V. Garcia, E. Debreuve, and M. Barlaud, "Fast k nearest neighbor search using gpu," in *Proceedings of Computer Vision and Pattern Recognition Workshops*, June 2008, pp. 1–6.

[3] T. D. R. Hartley, U. Catalyurek, A. Ruiz, F. Igual, R. Mayo, and M. Ujaldon, "Biomedical image analysis on a cooperative cluster of gpus and multicores," in *ICS '08: Proceedings of the 22nd annual international conference on Supercomputing*. New York, NY, USA: ACM, 2008, pp. 15–25.

[4] M. C. Schatz, C. Trapnell, A. L. Delcher, and A. Varshney, "High-throughput sequence alignment using graphics processing units," *BMC Bioinformatics*, vol. 8, no. 474, 2007.

[5] J. E. Stone, J. C. Phillips, P. L. Freddolino, D. J. Hardy, L. G. Trabuco, and K. Schulten, "Accelerating molecular modeling applications with graphics processors," *Journal of Computational Chemistry*, vol. 28, no. 16, pp. 2618 – 2640, 2007.

[6] J. A. van Meel, A. Arnold, D. Frenkel, S. F. P. Zwart, and R. G. Belleman, "Harvesting graphics power for md simulations," *Molecular Simulation*, vol. 34, pp. 259–266, 2008.

[7] A. Tasora, D. Negrut, and M. Anitescu, "Large-scale parallel multibody dynamics with frictional contact on the graphical processing unit," *Journal of Multibody Dynamics*, 2008, to appear.

[8] I. S. Ufimtsev and T. J. Martínez, "Quantum chemistry on graphical processing units. 1. strategies for two-electron integral evaluation," *Journal of Chemical Theory and Computation*, vol. 4, no. 2, pp. 222 –231, 2008.

[9] *CUDA Programming Guide, Version 2.0*, NVIDIA, June 2008.

[10] *CUDA CUBLAS Library, Version 2.0*, NVIDIA, March 2008.

[11] *CUDA CUFFT Library, Version 2.0*, NVIDIA, April 2008.

[12] D. L. Donoho, "For most large underdetermined systems of linear equations the minimal ℓ_1-norm solution is also the sparsest solution," *Communications in Pure and Applied Mathematics*, vol. 59, pp. 797–829, 2006.

[13] ——, "For most large underdetermined systems of linear equations the minimal ℓ_1-norm near-solution is also the sparsest near-solution," *Communications in Pure and Applied Mathematics*, vol. 59, pp. 907–934, 2006.

[14] E. Candès and T. Tao, "Near-optimal signal recovery from random projections and universal encoding strategies," October 2004.

[15] E. Candès, J. Romberg, and T. Tao, "Signal recovery from incomplete and inaccurate information," *Communications in Pure and Applied Mathematics*, vol. 59, no. 8, pp. 1207–1223, 2005.

[16] D. L. Donoho, "Compressed sensing," *IEEE Trans. Inform. Theory*, vol. 52, no. 4, pp. 1289–1306, Apr. 2006.

[17] E. J. Candès, "Compressive sampling," in *Proceedings of the International Congress of Mathematicians, Madrid*, 2006.

[18] P. L. Combettes and V. R. Wajs, "Signal recovery by proximal forward-backward splitting," *Multiscale Modeling and Simulation*, vol. 4, no. 4, pp. 1168–1200, 2005.

[19] M. A. T. Figueiredo and R. D. Nowak, "An EM algorithm for wavelet-based image restoration," *IEEE Transactions on Image Processing*, vol. 12, pp. 906–916, 2003.

[20] Y. Nesterov, "Gradient methods for minimizing composite objective function," CORE, Catholic University of Louvain, CORE Discussion Paper 2007/76, September 2007.

[21] A. Beck and M. Teboulle, "A fast iterative shrinkage-threshold algorithm for linear inverse problems," Technion-Israel Institute of Technology, Technical Report, July 2008.

[22] Y. Nesterov, "A method for unconstrained convex problem with the rate of convergence $o(1/k^2)$," *Doklady AN SSSR*, vol. 269, pp. 543–547, 1983.

[23] ——, *Introductory Lectures on Convex Optimization: A Basic Course*. Kluwer Academic Publishers, 2004.

[24] E. T. Hale, W. Yin, and Y. Zhang, "A fixed-point continuation method for ℓ_1-regularized minimization with applications to compressed sensing," CAAM, Rice University, CAAM Technical Report TR07-07, May 2007.

[25] M. A. Saunders, "PDCO: primal-dual interior-point method for convex objectives," Systems Optimization Laboratory, Stanford University, Tech. Rep., November 2002.

[26] S.-J. Kim, K. Koh, M. Lustig, S. Boyd, and D. Gorinevsky, "A method for large-scale ℓ_1-regularized least squares problems with applications in signal processing and statistics," Electrical Engineering Department, Stanford University, Technical Report, February 2007.

[27] E. Candès and J. Romberg, "ℓ_1-MAGIC: Recovery of sparse signals via convex programming," California Institute of Technology, Tech. Rep., October 2005.

[28] M. A. T. Figueiredo, R. D. Nowak, and S. J. Wright, "Gradient projection for sparse reconstruction: Application to compressed sensing and other inverse problems," *IEEE Journal on Selected Topics in Signal Processing*, vol. 1, no. 4, pp. 586–597, December 2007.

[29] S. J. Wright, R. D. Nowak, and M. A. T. Figueiredo, "Sparse reconstruction by separable approximation," in *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, March-April 2008.

[30] J. Barzilai and J. M. Borwein, "Two-point step size gradient methods," *IMA Journal of Numerical Analysis*, vol. 8, pp. 141–148, 1988.

[31] L. Rudin, S. Osher, and E. Fatemi, "Nonlinear total variation based noise removal algorithms," *Physica D*, vol. 60, pp. 259–268, 1992.

[32] M. Zhu and T. F. Chan, "An efficient primal-dual hybrid gradient algorithm for total variation image restoration," Mathematics Department, UCLA, CAM Report 08-34, May 2008.

[33] M. Zhu, S. J. Wright, and T. F. Chan, "Duality-based algorithms for total variation image restoration," Mathematics Department, UCLA, CAM Report 08-33, May 2008.

[34] T. F. Chan, G. H. Golub, and P. Mulet, "A nonlinear primal-dual method for total variation based image restoration," *SIAM Journal of Scientific Computing*, vol. 20, pp. 1964–1977, 1999.

[35] NVIDIA, *CUDA Visual Profiler 1.0*, June 2008.