On Optimizing Machine Learning Workloads via Kernel Fusion

Arash Ashari *
Department of Computer Science and Engineering, The Ohio State University, Columbus, OH, USA
ashari@cse.ohio-state.edu

Shirish Tatikonda
Matthias Boehm
Berthold Reinwald
IBM Research – Almaden, San Jose, CA, USA
statiko,mboehm, reinwald@us.ibm.com

Keith Campbell
John Keenleyside
Hardware Acceleration Laboratory, IBM, Markham, ON, Canada
keithc,keenley@ca.ibm.com

P. Sadayappan
Department of Computer Science and Engineering, The Ohio State University, Columbus, OH, USA
saday@cse.ohio-state.edu

Abstract

Exploitation of parallel architectures has become critical to scalable machine learning (ML). Since a wide range of ML algorithms employ linear algebraic operators, GPUs with BLAS libraries are a natural choice for such an exploitation. Two approaches are commonly pursued: (i) developing specific GPU accelerated implementations of complete ML algorithms; and (ii) developing GPU kernels for primitive linear algebraic operators like matrix-vector multiplication, which are then used in developing ML algorithms. This paper extends the latter approach by developing *fused* kernels for a combination of primitive operators that are commonly found in popular ML algorithms. We identify the generic pattern of computation $(\alpha * X^T \times (v \odot (X \times y)) + \beta * z)$ and its various instantiations. We develop a fused kernel to optimize this computation on GPUs – with specialized techniques to handle both *sparse* and *dense* matrices. This approach not only reduces the cost of data loads due to improved temporal locality but also enables other optimizations like coarsening and hierarchical aggregation of partial results. We also present an analytical model that considers input data characteristics and available GPU resources to estimate near-optimal settings for kernel launch parameters. The proposed approach provides speedups ranging from $2\times$ to $67\times$ for different instances of the generic pattern compared to launching multiple operator-level kernels using GPU accelerated libraries. We conclude by demonstrating the effectiveness of the approach in improving end-to-end performance on an entire ML algorithm.

Categories and Subject Descriptors D.1.3 [*Concurrent Programming*]: Parallel Programming; I.2.m [*Artificial Intelligence*]: Miscellaneous—Machine Learning

Keywords Machine Learning, GPU, Fused Kernel, Sparse, Dense

* This work was done during an internship at IBM Research – Almaden.

PPoPP'15, February 7–11, 2015, San Francisco, CA, USA

http://dx.doi.org/10.1145/2688500.2688521

1. Introduction

The recent surge in the amount of data collected for processing and analysis has sparked a widespread use of machine learning (ML) across a variety of application domains. ML is typically an iterative process and involves complex mathematical and statistical computations. Due to the sheer volume of data as well as the complexity of analysis, it is critical to exploit parallelism for enabling scalable and efficient machine learning.

Current developments in the field of computer architecture have shifted the industry focus from frequency scaling to the use of parallelism, leading to the advent of multi-core and many-core architectures. Massively parallel many-core GPUs are particularly interesting for ML, since a majority of its computations involve linear algebraic operations, which are highly optimized on GPUs. With a memory bandwidth exceeding 250GB/s and thousands of cores, GPUs are also attractive for memory-bound workloads like repeated matrix-vector multiplications. Furthermore, open standards for parallel programming like OpenCL and NVIDIA's CUDA have made it easier to exploit GPUs in applications other than graphics that are traditionally handled by CPUs [22].

There are two broad approaches for accelerating ML workloads using GPUs. The first approach is to hand-craft implementations as GPU kernels targeting specific algorithms, such as Support Vector Machines [8], neural networks [10], and decision trees [33]. These implementations are often packaged in the form of general-purpose libraries [4, 6, 7, 11, 25, 26] for the ease of use and adoption. However, such libraries of pre-canned implementations are not optimal for varying data characteristics and parameter settings. They are also not flexible for customizing existing algorithms and for developing new algorithms.

In contrast, the second approach uses primitive GPU-accelerated operators in composing complex ML algorithms. For example, a GPU implementation for linear regression can easily be realized by stitching together a sequence of GPU kernel invocations to dense and sparse matrix libraries, such as NVIDIA's cuBLAS[12] and cuSPARSE[15] (see Listing 1). There also exist other libraries of primitives such as cuDNN that are targeted at a particular class of ML algorithms, such as deep neural networks. We extend this direction of GPU exploitation by identifying and thereby developing a *fused* kernel for a combination of primitive linear algebraic operators. In particular, we identify the following generic pattern of computation used in a variety of ML algorithms:

$$w = \alpha * X^T \times (v \odot (X \times y)) + \beta * z \quad (1)$$

where α and β are scalars, w, y, and z are vectors, and X is a matrix that can be either sparse or dense. $\odot$ denotes element-wise multiplication, and $\times$ represents matrix multiplication.

Table 1. Example instantiations of the generic pattern from Equation 1, and their presence in ML algorithms.

Pattern Instantiation	LR	GLM	LogReg	SVM	HITS
$\alpha * X^T \times y$	✓	✓	✓	✓	✓
$X^T \times (X \times y)$	✓	✓		✓	✓
$X^T \times (v \odot (X \times y))$		✓	✓		
$X^T \times (X \times y) + \beta * z$	✓			✓	
$X^T \times (v \odot (X \times y)) + \beta * z$			✓		

Table 2. Breakdown of single-threaded CPU compute time (as percentages) for Linear Regression Conjugate Gradient.

Data set	Pattern	BLAS-Level 1	Total
KDD 2010 [19, 34]	82.9%	16.9%	99.8%
HIGGS [2]	99.4%	0.1%	99.5%

This particular pattern along with its various instantiations (shown in Table 1) is commonly found in a wide class of ML algorithms, such as linear regression (LR), generalized linear models [28] (GLM), binomial/multinomial logistic regression (LogReg, via trust region method [24]), support vector machines (SVM, training in the primal [9]), and Hubs and Authorities [23]. These instantiations are also one of the major performance bottlenecks (see Table 2). Approximately 82.9% (for KDD2010) or 99.4% (for HIGGS) of single-threaded CPU compute time [1] is spent in operations that are part of one or more of these patterns – thereby making the generic pattern a promising and viable target for GPU exploitation. The last column in the table highlights the total potential benefits from GPU acceleration, both by leveraging custom fused kernels as well as existing GPU accelerated libraries.

Challenges: Developing an efficient CUDA kernel for the identified pattern is challenging due to four reasons. First, for the general pattern, the input matrix X needs to be accessed both in row-major (for $X \times y$) and column-major (for $X^T \times (\cdot)$) order with the same underlying representation – storing X in both formats is inefficient. Second, inherent data-flow dependencies within the computation impede parallelism – for example, $X^T \times (\cdot)$ can be evaluated only after computing $(v \odot (X \times y))$. Third, it is non-trivial to realize an efficient result aggregation strategy that incurs minimal synchronization overhead, especially in the presence of CUDA's complex threading model with threads, warps, and blocks. Fourth, ensuring a balanced workload, maximizing thread occupancy, and realizing coalesced memory accesses in case of sparse matrices with different number of non-zeros across rows is difficult.

Contributions: We develop a fused kernel to deal with inherent data-flow dependencies in the computation. The entire computation is divided among sets of co-operating threads. Each set of threads operates on an independent portion of the input, and is responsible for generating the *final* partial result. This is done in a way such that the result of one operator is immediately fed into the next operator. Such a data parallel computation with a chain of operators eliminates the need for synchronization among sets, and thereby alleviating the problems posed by data-flow dependencies. We make the following contributions:

- We propose an optimized *fused* kernel that exploits *temporal locality* to reduce the number of loads while scanning the input.
- We develop a hierarchical aggregation strategy spanning the complete GPU memory hierarchy (registers, shared memory, global memory) that performs as many local aggregations as possible to reduce the synchronization overhead.
- We present a simple but effective analytical model to estimate kernel launch parameters that maximizes thread occupancy, minimizes atomic writes to global memory, and estimates near-optimal settings with minimal overhead.
- In the case of dense matrices, we propose a code generation technique that relies on *unrolling* to perform a majority of computations on GPU registers.
- Finally, we perform a detailed *pattern-level* as well as *end-to-end* experimental evaluation demonstrating the effectiveness of the proposed approach.

[1] For linear regression on KDD 2010 data set [19, 34], as measured on a commercial system *SystemML* [17], ignoring read-write time to disk and other architectural overheads.

2. Background and Related Work

Modern GPUs (such as NVIDIA GeForce GTX Titan) are equipped with thousands of cores (2,688), high bandwidth memory fabric (288GB/s), and complex memory hierarchy with global memory (6GB), texture memory, shared memory, L1 cache, and registers. GPU cores are organized into streaming multiprocessors or SMs (14 SMs with 192 cores each). GPU threads are grouped into thread *blocks* and each block executes on a single SM. Threads within the block share fast on-chip *shared memory* (48KB/SM), and can synchronize and share data with other threads in the same block. Thread blocks are further divided into *warps* of size 32 threads each. Threads within a warp execute instructions in Single-Instruction-Multiple-Data (SIMD) mode. Furthermore, a number of registers (64K) are also available per block. All blocks running on different SMs can access fast read-only texture memory (48KB) and a relatively slow global memory (6GB). Coalesced memory accesses among threads, number of accesses (read/write) to global memory, thread divergence within a warp, and load balance among the blocks are some of the factors that govern the overall performance from GPU acceleration.

Emergence of programmer-friendly low-level APIs such as Open Computing Language (OpenCL) [21] and NVIDIA's Compute Unified Device Architecture (CUDA) [13, 29] have made it easier for programmers to develop applications that are traditionally handled by CPUs, including ML. Exploitation of GPUs in applications is further made easy with the development of accelerated, high performance libraries. For example, NVIDIA's cuBLAS [12] and cuSPARSE [15] provide optimized GPU implementations for the complete standard BLAS library and basic linear algebra subroutines used for sparse matrices, respectively.

```
1  V = read($1); y = read($2);
2  eps = 0.001; tolerance = 0.000001;
3  r = -(t(V) %*% y); #cuBLAS/cuSPARSE: gemv/csrmv
4  p = -r;
5  nr2 = sum(r * r); #cuBLAS: nrm2
6  nr2_init = nr2; nr2_target = nr2 * tolerance ^ 2;
7  w = matrix(0, rows=ncol(V), cols=1);
8  max_iteration = 100; i = 0;
9  while(i < max_iteration & nr2 > nr2_target) {
10   q = ((t(V) %*% (V %*% p)) + eps * p);
11   # ... csrmv/gemv & axpy
12   alpha = nr2 / (t(p) %*% q); # ... dot
13   w = w + alpha * p; # ... axpy
14   old_nr2 = nr2;
15   r = r + alpha * q; # ... axpy
16   nr2 = sum(r * r); # ... nrm2
17   beta = nr2 / old_nr2;
18   p = -r + beta * p; # ... axpy & scal
19   i = i + 1;
20 }
21 write(w, "w");
```

Listing 1. Linear regression conjugate gradient algorithm.

Machine learning models are complex and expensive to build. Furthermore, searching for the right model and its parameters is often an iterative process. There have been several efforts in accelerating

ML workloads using GPUs. Specific hand-crafted implementations are developed for a variety of ML algorithms, such as artificial neural networks [10], support vector machines [8], decision trees and forests [33], deep belief networks [32], and k-Means [16]. These implementations are often packaged into libraries – such as BIDMach [6, 7], GPUMLib [25, 26], Theano [4], and Torch7 [11] – for easy adoption and usability. The recent BIDMach toolkit offers optimized implementations targeting both CPUs (powered by Intel MKL) and GPUs (custom kernels). It supports a number of supervised and unsupervised models, including regression, clustering, classification, and matrix factorization. It is built on a sister library called BIDMat [6, 7] that provides an efficient, interactive matrix layer. However, such libraries of pre-canned implementations are not flexible for customizing existing algorithms and for developing new algorithms. With that it imposes a huge development effort and it gets worse with new GPU generations because all of those implementations would need to be reconsidered.

In contrast, there exists another class of approaches that rely on GPU-accelerated primitive ML operators, such as matrix multiplication. ML algorithms are then composed by stitching together multiple primitive operators. For example, a GPU implementation of linear regression (shown in Listing 1) can be developed by using accelerated matrix-vector multiplication and vector-vector arithmetic operations. Accelerated operators are made available through a number of libraries, such as cuBLAS and cuSPARSE. Matrix Algebra on GPUs and Multicore Architecture (MAGMA) [1, 27] offers a library of CPU and GPU kernels for dense linear algebraic operations on heterogeneous GPU-based architectures. Similarly, High Performance Linear Algebra in R (HiPLAR) [18] delivers high performance linear algebra routines for the R platform. Recently, MATLAB enabled GPU computing through its parallel computing toolbox that supports GPU-enabled functions (e.g., fft, mtimes, and mldivide) and toolboxes (e.g., Image Processing Toolbox). There also exist accelerated libraries targeting a specific class of ML algorithms – for example, NVIDIA's cuDNN [14] provides a library of functions for building deep neural networks.

In this work, we extend the latter approach of GPU exploitation for ML by developing optimized fused kernels for a combination of primitive operators (see Equation 1), that is reusable for a wide class of ML algorithms. We now describe our approach of developing a *fused* kernel for the identified pattern of computation.

3. Fused Kernels

The identified pattern from Equation 1, along with its variants are commonly found in a variety of ML algorithms, and they also account for a significant fraction of CPU compute time (see Table 2). A direct approach for GPU exploitation is to launch separate kernels for individual operations. For example, $(X^T \times (X \times y))$ can be computed by invoking two BLAS Level 2 functions – one for matrix-vector multiplication $(X \times y)$ and another for transpose-matrix-vector multiplication $(X^T \times (\cdot))$. Such an approach is inefficient, especially because matrix-vector multiplication is memory-bound. Consider NVIDIA GeForce GTX Titan with 1.2 TFLOPs peak double precision performance. The bandwidth to global memory is 288 GB/sec (ECC off). Peak performance can only be achieved when *every* data load is used in at least 34 floating point operations. However in the case of matrix-vector multiplication, every element read from X is used exactly once, i.e., 1 compute/load. Therefore, launching multiple kernels for such memory-bound computations will result in low performance.

Moreover, the computation of $(X^T \times (\cdot))$ is typically more expensive than $X \times y$. For sparse matrices, this is due to uncoalesced memory accesses caused by the mismatch in the access pattern (column-major) and the underlying storage layout (row-major, for CSR representation). In the case of dense matrices, blocks of X can be read and kept in shared memory for future access. Although the reads from global memory can be coalesced due to regular indexing, the accesses to shared memory may cause memory bank conflicts, resulting in poor performance.

These shortcomings of the naïve approach motivate the need for a *fused* kernel, wherein the data is propagated through a chain of operators without explicitly storing the intermediate results in global memory. Such an approach greatly alleviates the problem of repeated data loads in memory-bound workloads by exploiting temporal locality. Consider $(X^T \times (X \times y))$, where X is sparse. Each row r of X is still loaded twice, once to compute $p[r]$:

$$p[r] = v[r] \times (X[r,:] \times y[:]), \tag{2}$$

and once again to compute the partial result of w:

$$w[:] = X[r,:]^T \times p[r] \tag{3}$$

However, if we ensure that the second load of $X[r,:]$ (to compute w) is performed by the same threads that previously used the row for computing p, due to temporal locality the second load will likely to be a cache hit. This decreases the overhead due to loads potentially by a factor of up to 2. Such a behavior can be guaranteed when the number of non-zeros per row is bounded by the cache size. Finally, the partial values of w computed by multiple threads spanning warps and blocks must be aggregated to obtain the final result. To this end, we propose a hierarchical aggregation strategy spanning registers, shared memory, and global memory.

In this work, we assume that the input matrix X fits in the device memory. This allows amortization of the cost of data transfer between the host and the device across multiple iterations of an ML algorithm (see Listing 1, for example). In situations where such an amortization is not feasible, the developed methods can easily be adapted to a streaming design for "out-of-core" computation.

3.1 Fused Kernel for Sparse Matrices

In this subsection, we present our methods when the input matrix X is sparse. We start by describing algorithms that tackle components of Equation 1, and subsequently combine them to perform the complete computation. For the sake of simplicity, assume that the number of columns in X is small, so that the partial result of w can be kept in shared memory. We later generalize the algorithm by relaxing this assumption.

The basic component of Equation 1 is $X^T \times p$. Although cuSPARSE [15] offers an API for this specific operation, it is very slow when compared to $X \times p$. NVIDIA suggests an explicit transposition of the matrix (using $csr2csc$ API), followed by a standard sparse matrix-vector multiplication. However, this is inefficient due to the high cost in transposing the matrix, and the need to maintain both X and X^T on the device. Note that both X and X^T are required for the computation. Instead, we propose to compute $w = X^T \times p$ in two steps: 1) *intra-block* computation and aggregation, and 2) *inter-block* aggregation.

Table 3. Notation

Symbol	Description
m	Number of rows in X
n	Number of columns in X
VS	Vector Size, Number of threads within a vector
NV	Number of vectors within a block
BS	Block Size, Number of threads within a block
C	Degree of coarsening, Number of rows processed by vectors
TL	Thread load

In the first step, we leverage the idea of CSR-vector [3], and partition a block of threads into sets of cooperating threads called *vectors*. The number of threads in a vector is denoted as *vector size* VS. The number of vectors within a single block is denoted NV. All threads in a vector work on the same row r simultaneously

Algorithm 1: $(X^T \times p)$ Kernel for Sparse Matrices

input : CSR matrix X: $(values, col_{idx}, row_{off})$, m (# of rows), n (# of columns), Vector: p, VS (# of threads per vector), C (coarscening factor)
output: Vector: w

```
1  begin
2      tid ← thread local ID;
3      (lid, vid) ← (tid % VS, tid / VS); // lane and vector ID
4      NV ← blockSize / VS; // # of vectors per block
5      row ← block_ID × NV + vid;
6      SD[1 : n] ← 0; // on shared memory
7      for c ← 1 to C do
8          if (row < m) then
9              start ← row_off[row]; // current row offset
10             end ← row_off[row + 1]; // next row offset
11             for (i = start + lid; i < end; i+ = VS) do
12                 SD[column_idx[i]] += values[i] × p[row];
13         row += (gridSize/VS);
14     synchronize(); // wait till all vectors reach here
15     for (i = tid; i < n; i += blockSize) do
16         atomicAdd(w[i], SD[i]);
```

Algorithm 2: Fused Kernel for Sparse Matrices

input : CSR matrix X: $(values, col_{idx}, row_{off})$, Vectors: y, v, and z, Scalars: m, n, VS, C, α and β
output: Vector: w

```
1  begin
2      Initialization // lines 2-6 Algorithm 1
3      for (i = thread global id; i < n; i+ = gridsize) do
4          atomicAdd(w[i], β × z[i]);
5      for c ← 1 to C do
6          if (row < m) then
7              start ← row_off[row]; // current row offset
8              end ← row_off[row + 1]; // next row offset
9              sum ← 0;
10             for (i = start + lid; i < end; i+ = VS) do
11                 sum += values[i] × y[col_idx[i]];
12             sum = intra_vector_reduce(sum) ×v[row];
13             for (i = start + lid; i < end; i+ = VS) do
14                 SD[column_idx[i]] += values[i] × sum;
15         row += (gridSize/VS);
16     synchronize(); // wait till all vectors reach here
17     for (i = tid; i < n; i += blockSize) do
18         atomicAdd(w[i], α × SD[i]);
```

to compute partial results $w[:] = w[:] + X[r,:]^T \times p[r]$. Also, a given vector of threads is responsible for processing a total of C rows, which is referred to as the *degree of coarsening*. Let us further denote the number of rows in X by m and the number of columns by n – see Table 3 for complete notation.

In the case of a sparse matrix, different rows may have different numbers of non-zeros – leading to highly irregular accesses and write conflicts over w while computing the partial results. When VS is smaller than the number of non-zeros in a row, a single thread within a vector itself may produce results for multiple elements in w. Also, each element in w may get updated simultaneously by threads from different vectors operating on different rows. Handling such write conflicts requires a careful strategy for aggregation. A simple choice is to let threads maintain a local version of the entire vector w in the registers. This however is redundant and inefficient use of registers. Observe that the partial results produced from an input row span different elements in w – i.e., there is no need for *intra-vector* aggregation on w, and therefore w can be shared among all threads within a vector. Partial results produced by multiple vectors across all blocks must be aggregated on the global memory using atomic operations. Since such atomic accesses are slow, we perform an additional level of pre-aggregation among vectors within a single block. To facilitate this, we keep w in shared memory (that is on the SM), and perform *inter-vector* or *intra-block* aggregations via atomic operations.

In the second step of computing $X^T \times p$, we perform the final *inter-block* aggregation on the global memory. Since multiple thread blocks may write to the same element of w, this access needs to be atomic as well. To reduce synchronization overhead, we increase the degree of coarsening C and the block size to their maximum possible values, while achieving the maximum possible occupancy. Further details on this are provided later in the context of parameter tuning (see Section 3.3). The entire method to compute $X^T \times p$ is summarized in Algorithm 1. Lines 7-13 perform the first step and Lines 15-16 handle the second step. The synchronization in Line 14 ensures that all vectors within a block are finished before the results are used in the final aggregation. We show in Section 4 that this method of computing $X^T \times p$ performs significantly better than the implementation in the cuSPARSE library.

We next extend the method to a larger computation: $w = X^T \times (X \times y)$. Each row in X must now be accessed twice to perform two matrix-vector multiplications. The main goal behind our technique is to reuse the rows r while computing $w[:] = X[r,:]^T \times p[r]$, where $p[r] = X[r,:] \times y[:]$. The data is first loaded during the computation of $p[r]$. It can be seen from the computation that the dot product $X[r,:] \times y[:]$ produces the *final value* of a single scalar $p[r]$. Subsequently, elements of $X[r,:]$ are scaled by $p[r]$ to calculate partial results $w[:]$. We compute $X^T \times (X \times y)$ also in three steps: *intra-vector* computation and aggregation, *inter-vector* aggregation, and *inter-block* aggregation. The last two steps are similar to Algorithm 1. In the first step, we compute the dot-product $p[r] = X[r,:] \times y[:]$ by leveraging the ideas of CSR-vector and segmented reduction [3]. Note that this step introduces an additional level of aggregation, i.e., at the *register-level*. Threads within a vector compute the partial results of $p[r]$ in registers, which are subsequently aggregated using the *shuffle* instruction available on NVIDIA Kepler architectures. Such a hierarchical strategy of aggregation spans registers (*intra-vector*), shared memory (*inter-vector*), and global memory (*inter-block*).

We further extend this method by performing the element-wise vector-vector multiplication, as in $w = X^T \times (v \odot (X \times y))$. As soon as the value of $p[r]$ is computed by a vector of threads, one of those threads performs the element-wise multiplication $v[r] * p[r]$. Finally, with respect to the complete pattern, the computation $\beta * z$ can be considered as an initialization step that is performed in the beginning, by all threads.

The complete fused kernel is shown in Algorithm 2. The for loop in Lines 3-4 takes care of $\beta * z$ and the loop in Lines 5-15 performs the rest of the computation. For each row, threads compute partial results of $p[r]$ in Lines 10-12, performs register-level *intra-vector* aggregation and computes $p[r] * v[r]$ in Line 12. Subsequently, the partial result of w is computed on the shared memory in Lines 13-14 (center portion of Figure 1). The synchronization in Line 16 marks the completion of *inter-vector* aggregation – shown as gray bars in Figure 1. Aggregation on the global memory to yield the final result is done in Lines 17-18.

A key aspect here concerns the atomic adds in Line 4 during the scalar computation with β. While it can be easily parallelized

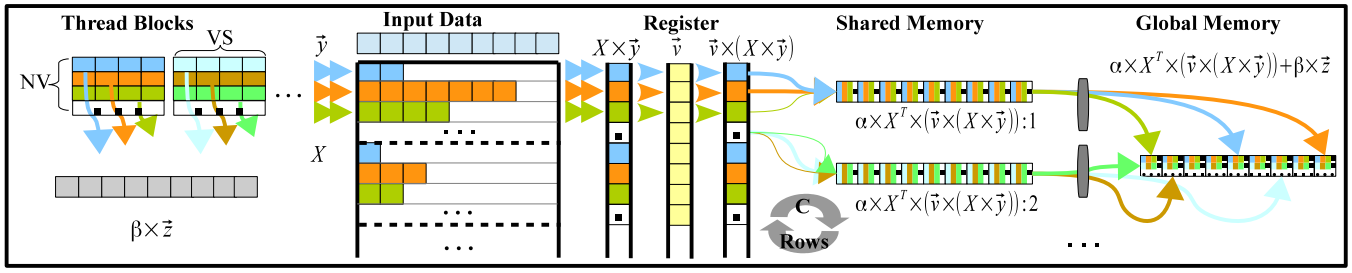

Figure 1. Different steps of fused kernel for sparse matrix; VS: vector size, NV: number of vectors in block, C: coarsening factor.

across thread blocks without atomic operations, it requires *inter-block* barrier synchronization to ensure that the subsequent computation starts only after $\beta * z$ is fully computed. Otherwise, it may lead to race conditions. CUDA, however, does not provide support for such *inter-block* barriers – to allow for flexibility in scheduling thread blocks. The only two alternatives here are either to use atomic adds as shown in Line 4 or to launch two kernels, one for computing $\beta * z$ and the other for rest of the computation.

Algorithm 2 is limited by the amount of available shared memory per SM on the GPU. Since the *inter-vector* aggregation is done using the shared memory, the number of columns n in X must be small so that w fits in the shared memory. The exact limit on n can be computed from two parameters VS and NV (see Section 3.3 for details). For a device with $48KB$ shared memory per SM, the limit on n is close to $6K$. Such matrices with limited number of columns are indeed common in many enterprise workloads [35].

It is easy to extend this method to handle matrices with large number of columns as well – for example, the $KDD2010$ [34] data set that we consider in our evaluation has 30M columns. Towards this end, we simply move the *inter-vector* aggregation from shared memory to global memory. The for loop in Lines 13-14 will now operate on global memory using atomic operations. Note that the *inter-block* aggregation in Lines 17-18 is no longer required. This modified strategy increases the synchronization overhead, especially when n is relatively small, due to large number of concurrent writes. However, as the shared memory limit gets mitigated, we can have as many active threads as the maximum number of concurrent threads on each SM. For example 64 active warps (2,048 threads) per SM with compute capability $\geq$ 3.5. By dynamically switching between warps when threads are stalled due memory access, such an access latency can be hidden. Furthermore, when n is very large, the data is likely to be sparse (e.g., social network data) and the likelihood of concurrent accesses to a single element of w is very small, thereby reducing the overhead due to atomic writes.

3.2 Fused Kernel for Dense Matrices

Our method for handling dense matrices is similar to Algorithm 2, in its spirit. Unlike the case of sparse matrices, we can leverage indexing since data accesses on dense data are very regular. We can use the local memory available on SMs (i.e., registers) to get higher access speeds than shared memory. However, the bank conflict rate increases by the order of number of warps in a block – since the number of shared memory banks on each SM is limited (32 in new NVIDIA Kepler GPUs). *Intra-vector* coarsening is more explicit and constant among rows, while it is implicit and row-dependent in the case of sparse data. Furthermore, each vector of threads processes C rows – a second level of coarsening.

Our method for the case of dense matrices is shown in Algorithm 3. Each row is processed by VS threads, where each thread processes TL (known as *thread load*) elements of the row. Variables l_y, l_X, and l_w in the algorithm refer to *registers* local to threads. In the first step, the elements of y are read by each vector once, and kept in registers l_y (Lines 4-5). Also, w is initialized with $\beta * z$ in Lines 6-7. Then for each row r, threads within the vector perform the following – read TL elements of X; multiply by the corresponding element of y; and compute the partial result of $X[r,:] \times y[:]$ (Lines 11-13). These partial results are then aggregated in Lines 14-22. If $VS \leq 32$, reduction is performed in a single step by all threads of the vector (Lines 14-15). Otherwise, it consists of two steps – an *intra-warp* reduction using registers via the *shuffle* instruction (similar to *intra-vector* reduction) in Line 19 followed by *inter-warp* reduction in Line 20. Appropriate synchronizations are performed to avoid any race conditions. The cell-wise multiplication between $p[r]$ and $v[r]$, is again done by a single thread (Lines 15 and 20). Each thread then scales the elements of $X[r,:]$ by $v[r] \times (X[r,:] \times y[:])$ (Lines 23-24), and aggregates them into a local register l_w – a partial result of w. Once all

Algorithm 3: Fused Kernel for Dense Matrices

input : Matrix: X, Vectors: y, v, and z, Scalars: m, n, VS, C, α, β, and TL (load per thread)
output: Vector: w

```
1  begin
2      Initialization // lines 2-5 Algorithm 1
3      l_w[1 : TL] ← 0;
4      for i ← 1 to TL do
5          l_y[i] = y[lid + i × VS];
6      for (i = thread global id; i < n; i+ = gridsize) do
7          atomicAdd(w[i], β × z[i]);
8      for c ← 1 to C do
9          if (row < m) then
10             sum ← 0;
11             for i ← 1 to TL do
12                 l_X[i] ← X[row, i × VS];
13                 sum += l_X[i] × l_y[i];
14             if (VS ≤ 32) then
15                 sum = intra_vector_reduce(sum) ×v[row];
16             else
17                 sum = intra_warp_reduce(sum);
18                 // wait till all warps reach here
19                 synchronize();
20                 sum = inter_warp_reduce(sum) ×v[row];
21                 // wait till inter-warp reduction finishes
22                 synchronize();
23             for i ← 1 to TL do
24                 l_w[i] += l_X[i] × sum;
25         row += (gridSize/VS);
26     for i ← 1 to TL do
27         atomicAdd(w[lid + i × VS], α × l_w[i]);
```

assigned rows are processed, threads within each vector propagate their partial results in l_w to global memory (Lines 26-27).

Note that elements of X, y and the partial results of w are kept in registers – l_X, l_y and l_w, respectively. Accesses to these registers are done via indexing. However, if the index value is unknown at compile time, CUDA forces these accesses to use global memory instead of registers, thereby significantly degrading the performance. To deal with this problem, we resort to an idea of *code generation*. Since the matrix dimensions and input parameters are known at the time of invoking a ML algorithm, we use a code generator to produce the kernel that uses explicit registers and performs loop-unrolling for Lines 4-5, 11-13, 23-24, and 26-27 in Algorithm 3. Listing 2 shows an example of generated kernel $mtmvm$, for producing $\alpha * X^T \times (v \odot (X \times y))$ when the inputs are: a dense matrix X of size $m \times 32$; $VS = 16$; and $TL = 2$. Line 17 in Listing 2 corresponds to Lines 4-5 in Algorithm 3, in which the for loop has been unrolled. Instead of register indexing, registers are accessed directly via explicit names, such as l_{y1} and l_{y2}. Note that the loop unrolling factor is equal to the thread load TL. This parameter can be chosen depending upon the number of available registers (see Section 3.3).

In Algorithm 3, we assume that the number of columns n is a multiple of the vector size VS. We make this assumption to avoid thread divergence inside a vector (and a warp, accordingly). When $n\%VS \neq 0$, we pad both matrix X and vector y with zero rows. In the worst case, we pad by only $VS - 1$ rows – hence, the cost of padding is negligible. This is done prior to launching the kernel.

```
1  __global__ void mtmvm_32_16_2(const double *X,
       const double *y, const double *v, const a,
       double *w) {
2   __shared__ volatile double sdata[16];
3   unsigned int tid = threadIdx.x;
4   unsigned int lid = tid & (15);
5   unsigned int vid = tid / 16;
6   unsigned int rowStart = blockIdx.x * NV + vid;
7   unsigned int rowEnd = rowStart + (gridDim.x * NV)
        * rowPerVector;
8   double sum, l_y1, l_y2, l_X1, l_X2, l_w1, l_w2;
9   if (tid < 16)
10    sdata[tid] = 0;
11  if (rowStart < rowDim) {
12    if (rowEnd > rowDim)
13      rowEnd = rowDim;
14    rowStart = rowStart * colDim + lid;
15    rowEnd = rowEnd * colDim + lid;
16    l_w1 = l_w2 = 0.0f;
17    l_y1 = y[lid]; l_y2 = y[lid + 16];
18    for (; rowStart < rowEnd; rowStart += (gridDim.x
          * NV) * colDim) {
19      l_X1 = X[rowStart]; sum = l_X1 * l_y1;
20      l_X2 = X[rowStart + 16]; sum += l_X2 * l_y2;
21      sum = interVectorReduce(sum);
22      if (lid == 0)
23        sdata[vid] = sum * v[rowStart/colDim];
24      sum = sdata[vid];
25      l_w1 += l_X1 * sum; l_w2 += l_X2 * sum;
26    }
27    r = r + lid;
28    atomicAdd(r, a * l_w1);
29    atomicAdd(r + 16, a * l_w2);
30  }
31 }
```

Listing 2. Generated kernel for a dense matrix of size $m \times 32$, $VS = 16$, and $TL = 2$.

The number of registers available on the GPU governs the maximum number of columns that can be handled by Algorithm 3. For a NVIDIA Kepler device with $64K$ registers per SM, the limit on n is close to $6K$. In order to process matrices with large n, one has to move all aggregations from registers to global memory. Since the X is dense, the number of concurrent writes to global memory is likely to be very high, thereby leading to serialization of writes. In such a scenario, we propose not to use the fused kernel, and instead, simply launch two separate cuBLAS Level 2 kernels. We note however that, in practice, it is not very common to encounter such dense data sets with large number of columns [35].

3.3 Parameter Tuning

Input parameters of proposed kernels have to be tuned appropriately, in order to realize best performance. We now present analytical models to determine best configurations for these parameters, by considering both input matrix characteristics, and available resource limits on the underlying GPU.

Parameters for a Sparse Kernel: Here, the main parameters are: *vector size* VS; *block size* BS; and *coarsening factor* C. Other parameters such as NV, number of vectors in a thread block, or grid size, can be determined accordingly.

VS determines number of cooperating threads that work on the same row(s) together. As we leverage the idea of segmented reduction and CSR-Vector in our kernel, we follow a known strategy to determine VS [3]. Let μ denote the average number of non-zero elements per row in a sparse matrix with a total number NNZ of non-zeros elements, i.e., $\mu = NNZ/m$. Then, VS is chosen from the set $\{2^0, ..., 2^5\}$ as follows:

$$VS = \begin{cases} 32 & \text{if } \mu > 32 \\ 2^i & \text{if } 2^{i+1} \geq \mu > 2^i, i \in [1, 4] \\ 1 & otherwise. \end{cases} \quad (4)$$

As the computation is memory-bound, we then determine BS in such a way that it maximizes the occupancy on the GPU. For this purpose, we consider the following limits of the device:

- Available register and shared memory per SM
- Maximum number of threads per block, and per SM
- Maximum number of active blocks
- Register and shared memory allocation granularity
- Maximum number of registers per thread
- Register and shared memory allocation units

For GPUs with compute capability ≥ 3.5, these limits are – (64K 32bit-registers and 48KB), (1,024 and 2,048), (8 blocks), (256 registers and 4 warps – per thread block), (256), and (256 and 256Bytes), respectively.

Our kernel requires 43 registers per thread (found via NVIDIA Visual Profiler [31]), and $(BS/VS + n) \times sizeof(precision)$ shared memory. For different values of BS from $\{1 \times 32, ..., 32 \times 32\}$, we calculate the number of concurrent warps, considering the number of registers and shared memory required. Exact value of BS is chosen to maximize the number of concurrent blocks, NW. This is similar to NVIDIA occupancy calculator [30].

To set the coarsening factor C, we aim to reduce the number of atomic write accesses to global memory. We set C so that all warps have maximal balanced workload, i.e.,

$$C = \left\lceil \frac{M}{NSM \times \frac{NW}{VS}} \right\rceil \quad (5)$$

where NSM is the number of available SMs on the GPU.

Parameters for a Dense Kernel: In the case of a dense kernel, C is set similarly. However, we first determine TL and BS – VS can be set accordingly. Here, we heavily use registers and to minimize the waste of register allocation we set BS to a size that is a multiple of the register allocation granularity; $\{1 \times 128, ..., 8 \times 128\}$. Furthermore, to minimize the overhead of *inter-vector* synchronization, we set BS to the minimum possible value; i.e. $BS = 128$. For setting TL, we profiled the dense kernel with different values of TL, and recorded the number of registers required in each

case. Our kernel requires 23 registers with $TL = 1$ and can handle up to $TL = 40$ (requiring 255 registers). Since register spilling severely degrades the performance with $TL > 40$, we only consider $TL \in \{1, ..., 40\}$. Having this information and GPU resource limits, we can calculate the number of concurrent warps NW. Furthermore, we also perform a refinement to exclude number of warp loads that are wasted by each vector. For example, with $BS = 128$, $TL = 2$ and $n = 200$, we have 1 wasted warp $\lfloor \frac{2\times128-200}{32} \rfloor$. While with $TL = 7$, there is no wasted warp per vector; $\lfloor \frac{7\times32-200}{32} \rfloor$. After setting TL based on the maximum number of concurrent warps, excluding wasted warps per vector, VS is set as:

$$VS = \begin{cases} BS & \text{if } \frac{n}{TL} > 32 \\ 2^i & \text{if } 2^i \geq \frac{n}{TL} > 2^{i-1}, i \in [1,5] \\ 1 & otherwise. \end{cases} \quad (6)$$

There is one exception: when the number of columns is less than the warp size, $n \leq 32$, we set $BS = 1024$ and $TL = 1$. In such a case, since we are not limited by the synchronization overhead, we can use the maximum possible thread block size. Furthermore, each thread only loads a single element of a row from the matrix, and hence, the use of a large number of threads helps in hiding the latency during data loads.

4. Experimental Results

We now empirically evaluate the performance of proposed methods using a combination of synthetic and real-world data sets. For this purpose, we use NVIDIA GeForce GTX Titan with 6GB global memory, compute capability 3.5, and CUDA driver version 6.0. This device is attached via PCIe-Gen3 (32GB/s) to a host with Intel core-i7 3.4 GHz CPU, 16GB memory, and 4 cores (8 hyper-threads). We compare the performance of our methods against a baseline approach of using GPU accelerated NVIDIA libraries cuBLAS and cuSPARSE to compute various instances of the generic pattern in Equation 1. We also consider other libraries like BIDMat [6] in our evaluation. Note that we do not evaluate the case of simple matrix-vector multiplication (both when X is sparse and dense). This is because cuBLAS and cuSPARSE libraries for this case already deliver optimized performance. Similarly, we do not consider $X^T \times y$, when X is dense.

4.1 Performance on Sparse Matrices

We begin by comparing the performance of our method for sparse matrices from subsection 3.1 against cuSPARSE and BIDMat [6]. BIDMat is an interactive matrix library that integrates CPU and GPU acceleration. BIDMat uses Intel MKL [20] for CPU acceleration and implements specialized kernels for GPU acceleration. We consider a randomly generated synthetic data set with number of rows in X is set to $500k$ and we vary the number of columns from 200 to 4,096. The sparsity is set to 0.01.

For the simple pattern $X^T \times y$, the observed speedups from our method are shown in Figure 2-top. On average, our proposed kernel is roughly $35\times$ faster than cuSPARSE, with highest speedups up to 67x in the lower end of the spectrum. The performance differences are primarily due to a difference in the number of global load transactions, as shown in Figure 2-bottom – note that y-axis is shown in *log* scale. Our method consistently performs less number of loads when compared to cuSPARSE across the board – on average, cuSPARSE performs $3.5\times$ more loads. This may be due to explicit construction of X^T and the use of semaphores (cuSPARSE is not open source). The performance of BIDMat is similar to cuSPARSE. Furthermore, the performance of our method is driven by the fact that data accesses are always performed in a coalesced manner, and the input vector y is always bound to texture memory, thereby improving accesses over y. By measuring the time to explicitly construct X^T and then to compute $X^T \times y$, we determine the number of iterations in ML algorithms required to amortize the transpose time into matrix-vector product computation – shown in the second X axis in Figure 2. High number of iterations in the figure indicate that explicitly constructing the transpose is an inefficient approach when dealing with sparse matrices.

For the pattern $X^T \times (X \times y)$, observed speedups for the synthetic data are shown in Figure 3. The benefits from our fused kernel are evident from the figure, and it outperforms the alternative methods for all matrix sizes. The average speedup is observed to be $9.28\times$, $14.66\times$, and $20.33\times$, against BIDMat-CPU (MKL with 8 hyper threads), BIDMat-GPU, and cuSPARSE, respectively. These speedups are primarily due to – improved data accesses via temporal locality; avoidance of materializing intermediate results through fused kernels; and finally, hierarchical aggregation strategy spanning registers, shared memory, and global memory.

Finally, Figure 4 shows the achieved speedups of the proposed kernel in computing the complete pattern $\alpha * X^T \times (v \odot (X \times y)) + \beta * z$. Since the whole computation is bottlenecked by $X^T \times (X \times y)$, we expect the performance differences to be similar or slightly better than those in Figure 3. On average, we observe speedup to be up to $13.41\times$, $19.62\times$, and $26.21\times$ against BIDMat-CPU (MKL with 8 hyper threads), BIDMat-GPU, cuBLAS/cuSPARSE, respectively. Note that, cuBLAS is used here to perform (Level 1) vector-vector computations.

Table 4. Execution time (in milliseconds) of the proposed kernel against cuBLAS/cuSPARSE on *KDD 2010* data set.

Pattern	Proposed	cuBLAS/cuSPARSE
$X^T \times y$	50.5	5552.1
$X^T \times (X \times y)$	78.3	5683.1
$\alpha * X^T \times (v \odot (X \times y)) + \beta * z$	85.2	5704.1

As noted in Section 3, when the number of columns in the input matrix is very large, we move all intra-block aggregations from shared memory to global memory. To measure the impact of such a choice, we consider an ultra-sparse real-world data set KDD2010 [19, 34] with 15,009,374 rows, 29,890,095 columns, and 423,865,484 non-zeros. Table 4 shows the execution time of our proposed approach against a method using cuSPARSE/cuBLAS libraries – the numbers shown are in milliseconds. Our method can efficiently handle data sets with large n as well – with more than two orders of magnitude improvement in case of $X^T \times y$, and a 66-fold speedup when computing the full pattern. When n is very large, the data set is likely to be sparse, leading to a reduced number of concurrent access to a single element of w on global memory. Therefore, the impact of concurrent atomic writes to global memory is also likely to be low.

4.2 Performance on Dense Matrices

We conducted a similar experiment to evaluate our dense fused kernel. Figure 5 shows that our method outperforms all the competing methods on dense matrices. On average, the observed speedups are $15.33\times$, $2.18\times$, and $4.27\times$ over BIDMat-CPU (MKL with 8 hyper threads), BIDMat-GPU, and cuBLAS. In general, we expect that our fused kernel for dense matrices does not reach the same level of improvement as we observed on sparse matrices. This is consistent with the performance differences observed between $X \times y$ and $X^T \times y$ on dense and sparse matrices using cuBLAS/cuSPARSE, indicating that most of the gain we achieve comes from loading X only once. Moreover, as the figures shows, MKL (CPU) works better on sparse matrices compared to BIDMach-GPU and cuSPARSE, while it performs worse on dense matrices since regular accesses brings more improvements on the GPU side. In Figure 5, we use matrices with number of columns up to $2K$. For $m > 2K$,

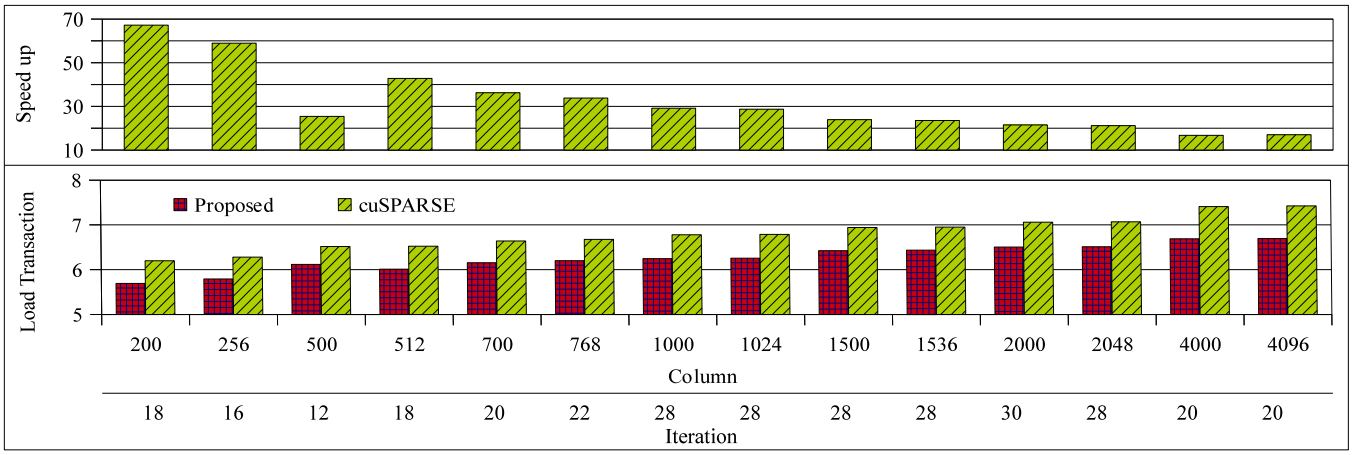

Figure 2. For $X^T \times y$ when X has 500k rows and sparsity 0.01: (Top) achieved speedups against cuSPARSE; (Bottom) number of load transactions (log_{10} scale); and *Iter* # of iterations required for transpose time to be amortized into product computation.

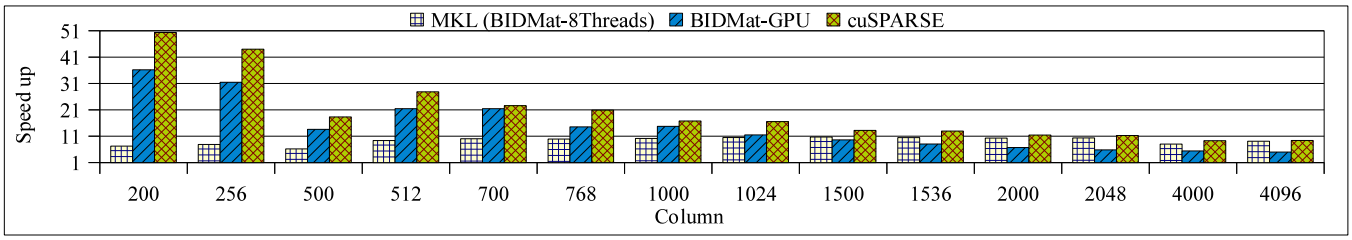

Figure 3. For $X^T \times (X \times y)$ when X has 500k rows and sparsity 0.01: Speedup achieved by the proposed kernel against cuSPARSE, BIDMat-GPU, and BIDMat-CPU (MKL-8Threads).

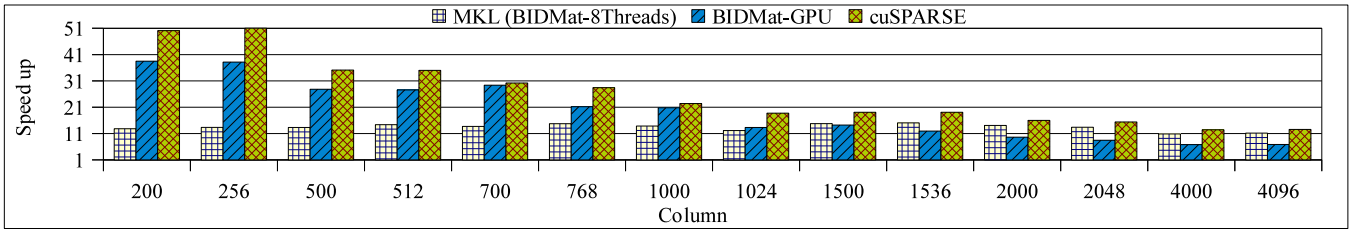

Figure 4. For $\alpha * X^T \times (v \odot (X \times y)) + \beta * z$ when X has 500k rows and sparsity 0.01: Speedup achieved by the proposed kernel against cuBLAS/cuSPARSE, BIDMat-GPU, and BIDMat-CPU (MKL-8Threads).

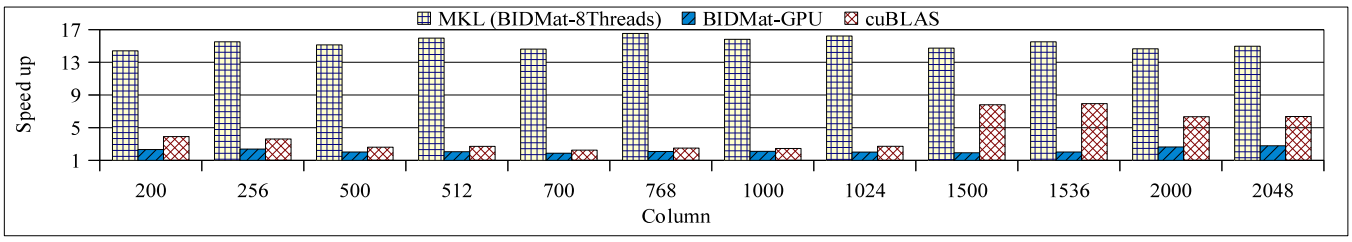

Figure 5. For $X^T \times (X \times y)$ when X is dense with 500k rows: Speedup achieved by the proposed kernel against cuBLAS, BIDMat-GPU, and BIDMat-CPU (MKL-8Threads).

the matrix does not fit in device memory anymore. We later discuss ideas for out of core computation. As stated earlier, we perform code generation in the case of dense matrices. In the generated kernel, we unroll computational loops, load X and y to registers once, and use them as often as required. Listing 2 shows an example generated kernel. We observed that the time spent in code generation is negligible when compared to the actual computation time. We also note that the performance of proposed kernel on the full pattern is

similar to that shown in Figure 5, since the majority of time while computing the full pattern is spent in $X^T \times (X \times y)$.

4.3 Parameter Tuning

We now evaluate the effectiveness of our analytical model that estimates the kernel launch parameter settings. The goal our model is to maximize occupancy while moving the overhead of atomic accesses from global memory to shared memory and to registers, as much as possible. The model incurs minimal performance overhead, since the parameter exploration is done in a limited search space. VS is chosen based on Equations 4 and 6, while the block size is selected from the set $\{2^5, ..., 2^{10}\}$. The number of rows handled by a set of cooperating threads (vector) is set to possible numbers around what our model selects; grid size is also chosen accordingly. The entire search space, on average, consists of about 1,200 different settings – see Figure 6. Note that y axis is shown as $1/Time$, so that the peaks with small time are clearly visible. For a sparse matrix of size $500k \times 1k$, the figure shows that the performance difference in computing $X^T \times (X \times y)$ using the optimal setting and the configuration chosen by our model is less than 2%. This difference is also less than 0.1% of the performance difference between the best and the worst settings. Furthermore, the configuration chosen by the model is one of the best 1% settings in terms of execution time.

This sparse kernel requires 43 registers per thread, selects $VS = 8, BS = 640$, and uses $(BS/VS+n) \times sizeof(precision)$ $= 8{,}832B$ shared memory per thread block. It then sets number of blocks to 28, where each vector takes care of 223 rows of the matrix. For a dense kernel, the thread load TL can range from 1 (requiring 24 registers) to 40 (requiring 255 registers). Using TL larger than 40 results in register spilling and in poor performance. We set VS so that each thread handles a maximum of 40 elements in a row. Corresponding 40-way unrolling in the computational loop of the kernel is handled by the code generator.

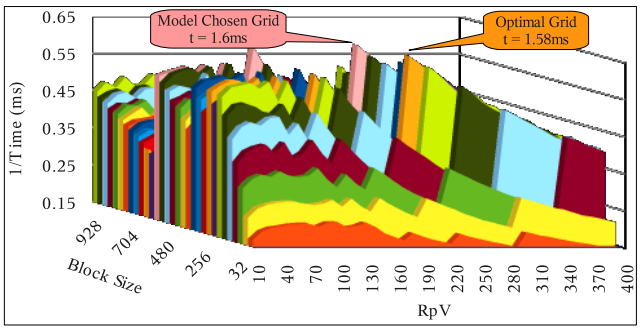

Figure 6. Inverse of time (1/ms) in computing $X^T \times (X \times y)$ using the proposed kernel with different plans on a $500k \times 1k$ sparse matrix with sparsity 0.01, and $VS = 8$. *RpV*: number of rows is processed by each vector.

4.4 End-to-end Analysis

Finally, we evaluate the end-to-end performance achieved by the proposed method in executing an entire machine learning algorithm. We consider linear regression, shown in Listing 1. We conduct this experiment with two real-world data sets – KDD2010 [19, 34] (sparse) as well as HIGGS [2] (dense). HIGGS data set is a dense matrix with $11,000,000$ rows, 28 columns, and $283,685,620$ non-zeros. We implemented the GPU accelerated linear regression algorithm by stitching together a list of CUDA kernel invocations. Here, we implement two versions – one with purely cuBLAS/cuSPARSE kernels (denoted $cu - end2end$), and the other with invocations to our fused kernel ($ours - end2end$). Since data transfer between the host and the device is limited by PCIe bus, it can be a potential bottleneck for large data sets. Therefore, we also take into account the time spent in data transfer to measure the ultimate benefits from our proposed method. Table 5 shows the speedups achieved by $ours - end2end$ against $cu - end2end$. The time to transfer KDD data set from host to device is observed to be 939 milliseconds. This time is amortized over ML iterations – Table 5 also shows the number of iterations executed on both data sets. Overall, we achieve up to $9\times$ end-to-end speedup when compared to the baseline strategy. This result demonstrates that the proposed method not only improves the performance of individual patterns of computation but also provides significant speedup for the entire algorithm.

Table 5. Speedup achieved by proposed (and cuBLAS/cuSPARSE) kernels against pure cuBLAS/cuSPARSE kernels in running Linear Regression Conjugate Gradient algorithm.

Data set	*HIGGS*	*KDD 2010*
Total Speedup	4.8×	9×
Number of ML iterations	32	100

Table 6. Speedup of GPU-enabled *SystemML* against its CPU version in running Linear Regression Conjugate Gradient algorithm.

Data set	*HIGGS*	*KDD 2010*
Total Speedup	1.2×	1.9×
Fused Kernel Speedup	11.2×	4.1×
Number of ML Iterations	32	100

We are now in the process of integrating the fused kernel into a commercial system for large-scale ML, namely *SystemML* [5, 17]. Such an integration requires three main components – (i) a cost model that helps in scheduling operations between the host and the device; (ii) a GPU memory manager that monitors the data transfer activities; and finally (iii) backend GPU kernels and APIs. Contributions from this paper are part of the final component of GPU kernels. We have developed preliminary versions of the first and second components. For example, the memory manager component is designed to perform the following tasks – a) allocate memory if it is not already allocated on the device; b) if there is not enough memory available on the device, perform necessary evictions to make room for incoming data; c) deallocate unnecessary variables/data and mark them for later reuse; d) maintain consistency between the copies of data maintained on CPU and GPU through appropriate synchronizations; and e) perform necessary data transformations to account for the differences in the data structures used on CPU and GPU. For example, *SystemML* represents a sparse matrix as an array of sparse rows on CPU, whereas the same matrix is represented in CSR format on the device. Furthermore, *SystemML* is implemented in Java. Therefore, one has to first transfer data from JVM heap space into native space via JNI, before it can be copied to the device. Such data transformations and JNI data transfers can potentially impact the performance. The cost model and the memory manager must be designed in such a way that the impact is minimized as much as possible. With the current preliminary versions of cost model and memory manager, the observed end-to-end speedups from Java including all the above mentioned overheads are shown in Table 6. It is important to note that the overall speedup from the fused kernel alone is more than $10\times$ in the case of HIGGS data set and $4\times$ in the case of KDD2010 data set. Reduced end-to-end speedups when compared to the ones in Table 5 point to the inefficiencies in our current memory manager and data transformations – this marks our ongoing and future research direction.

5. Conclusion and Future Work

GPUs provide massive parallelism for compute intensive workloads. This paper describes the exploitation of GPUs to accelerate a wide range of ML algorithms, such as regression and SVMs. By analyzing the characteristics of various ML algorithms, we observed that they share the common compute pattern $\alpha * X^T \times (v \odot (X \times y)) + \beta * z$ that manifests itself in various instantiations. Computing the pattern using primitive kernels in existing GPU libraries dominates the compute time of these algorithms. We developed generic, fused GPU kernels that are optimized for different data characteristics. The fused kernels minimize data transfer, optimize coarsening, and minimize synchronization during aggregation of partial results. These fused kernels provides speedups ranging from $2\times$ to $67\times$ for different instances of the generic pattern compared to running existing kernels. Furthermore, we describe an experimental result for an end-to-end GPU accelerated ML system that transparently selects our fused GPU kernel.

In the context of larger computations, although parts of the computation could be done on GPUs, it may not always be beneficial given the overhead of data transfer, and GPU memory limitations. Future work includes the development of a cost model that based on a complete system profile decides on hybrid executions involving CPUs and GPUs. A comprehensive system also needs to factor in data format conversions and GPU memory management including data replacement strategies in the face of iterative computations.

Acknowledgments

We would like to thank Alexandre Evfimievski for his initial ideas on identifying common computational patterns across different machine learning algorithms.

References

[1] E. Agullo, J. Demmel, J. Dongarra, B. Hadri, J. Kurzak, J. Langou, H. Ltaief, P. Luszczek, and S. Tomov. Numerical Linear Algebra on Emerging Architectures: The PLASMA and MAGMA Projects. *Journal of Physics: Conference Series*, 180(1):012037, 2009.

[2] P. Baldi, P. Sadowski, and D. Whiteson. Searching for Exotic Particles in High-Energy Physics with Deep Learning. *Nature communications*, 5, 2014.

[3] N. Bell and M. Garland. Implementing Sparse Matrix-Vector Multiplication on Throughput-Oriented Processors. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, page 18. ACM, 2009.

[4] J. Bergstra, O. Breuleux, F. Bastien, P. Lamblin, R. Pascanu, G. Desjardins, J. Turian, D. Warde-Farley, and Y. Bengio. Theano: a CPU and GPU Math Expression Compiler. In *Proceedings of the Python for scientific computing conference (SciPy)*, volume 4, page 3, 2010.

[5] M. Boehm, S. Tatikonda, B. Reinwald, P. Sen, Y. Tian, D. Burdick, and S. Vaithyanathan. Hybrid Parallelization Strategies for Large-Scale Machine Learning in SystemML. *Proceedings of the VLDB Endowment*, 7(7):553–564, 2014.

[6] J. Canny and H. Zhao. Big Data Analytics with Small Footprint: Squaring the Cloud. In *Proceedings of the 19th international conference on Knowledge discovery and data mining*, pages 95–103, 2013.

[7] J. Canny and H. Zhao. BIDMach: Large-Scale Learning with Zero Memory Allocation. In *BigLearning, NIPS Workshop*, 2013.

[8] B. Catanzaro, N. Sundaram, and K. Keutzer. Fast Support Vector Machine Training and Classification on Graphics Processors. In *Proceedings of the 25th international conference on Machine learning*, pages 104–111. ACM, 2008.

[9] O. Chapelle. Training a Support Vector Machine in the Primal. *Neural Computation*, 19(5):1155–1178, 2007.

[10] A. Coates, B. Huval, T. Wang, D. Wu, B. Catanzaro, and N. Andrew. Deep Learning with COTS HPC Systems. In *Proceedings of the 30th International Conference on Machine Learning*, pages 1337–1345, 2013.

[11] R. Collobert, K. Kavukcuoglu, and C. Farabet. Torch7: A Matlab-like Environment for Machine Learning. In *BigLearning, NIPS Workshop*, 2011.

[12] cuBLAS. The NVIDIA CUDA Basic Linear Algebra Subroutines Library. URL `https://developer.nvidia.com/cublas`.

[13] CUDA. A Parallel Computing Platform and Programming Model Invented by NVIDIA. URL `http://www.nvidia.com/object/cuda_home_new.html`.

[14] cuDNN. The NVIDIA CUDA Library of Primitives for Deep Neural Networks. URL `https://developer.nvidia.com/cuDNN`.

[15] cuSPARSE. The NVIDIA CUDA Sparse Matrix Library. URL `https://developer.nvidia.com/cusparse`.

[16] R. Farivar, D. Rebolledo, E. Chan, and R. H. Campbell. A Parallel Implementation of K-Means Clustering on GPUs. In *PDPTA*, pages 340–345, 2008.

[17] A. Ghoting, R. Krishnamurthy, E. Pednault, B. Reinwald, V. Sindhwani, S. Tatikonda, Y. Tian, and S. Vaithyanathan. SystemML: Declarative Machine Learning on MapReduce. In *IEEE 27th International Conference on Data Engineering*, pages 231–242. IEEE, 2011.

[18] HiPLAR. High Performance Linear Algebra in R. URL `http://hiplar.org`.

[19] C.-H. Ho and C.-J. Lin. Large-Scale Linear Support Vector Regression. *The Journal of Machine Learning Research*, 13(1):3323–3348, 2012.

[20] Intel. Math Kernel Library. URL `https://software.intel.com/en-us/intel-mkl`.

[21] Khronos OpenCL Working Group. *The OpenCL Specification, version 1.0.29*, December 2008.

[22] D. B. Kirk and W. H. Wen-mei. *Programming Massively Parallel Processors: a Hands-on Approach*. Newnes, 2012.

[23] J. M. Kleinberg. Authoritative Sources in a Hyperlinked Environment. *Journal of the ACM (JACM)*, 46(5):604–632, 1999.

[24] C.-J. Lin, R. C. Weng, and S. S. Keerthi. Trust Region Newton Method for Logistic Regression. *Journal of Machine Learning Research*, 9: 627–650, 2008.

[25] N. Lopes and B. Ribeiro. GPUMLib: An Efficient Open-Source GPU Machine Learning Library. *International Journal of Computer Information Systems and Industrial Management Applications*, 3:355–362, 2011.

[26] N. Lopes, B. Ribeiro, and R. Quintas. GPUMLib: a New Library to Combine Machine Learning Algorithms with Graphics Processing Units. In *Hybrid Intelligent Systems (HIS), 2010 10th International Conference on*, pages 229–232. IEEE, 2010.

[27] MAGMA. Matrix Algebra on GPU and Multicore Architectures. URL `http://icl.cs.utk.edu/magma`.

[28] P. McCullagh. Generalized Linear Models. *European Journal of Operational Research*, 16(3):285–292, 1984.

[29] J. Nickolls, I. Buck, M. Garland, and K. Skadron. Scalable Parallel Programming with CUDA. *Queue*, 6(2):40–53, 2008.

[30] NVIDIA. CUDA GPU Occupancy Calculator. URL `http://developer.download.nvidia.com/compute/cuda/CUDA_Occupancy_calculator.xls`.

[31] NVVP. NVIDIA Visual Profiler. URL `https://developer.nvidia.com/nvidia-visual-profiler`.

[32] R. Raina, A. Madhavan, and A. Y. Ng. Large-Scale Deep Unsupervised Learning Using Graphics Processors. In *International Conference on Machine Learning*, volume 9, pages 873–880, 2009.

[33] T. Sharp. Implementing Decision Trees and Forests on a GPU. In *Computer Vision–ECCV 2008*, pages 595–608. Springer, 2008.

[34] J. Stamper, A. Niculescu-Mizil, S. Ritter, G. Gordon, and K. Koedinger. Algebra I 2008-2009. Challenge Data Set from KDD Cup 2010 Educational Data Mining Challenge, 2013. URL `http://pslcdatashop.web.cmu.edu/KDDCup/downloads.jsp`.

[35] C. Zhang, A. Kumar, and C. Ré. Materialization Optimizations for Feature Selection Workloads. In *SIGMOD*, 2014.