Accelerating the Big Data Analytics Suite

Pierre Carrier
Applications Engineering
Hewlett-Packard Enterprise
Bloomington, MN, USA
Pierre.Carrier@hpe.com

Scott Moe[1]
HPC Software Solutions
Advanced Micro Devices, Inc.
Bellevue, WA, USA
scott.moeuw@gmail.com

Colin Wahl
Development Engineering
Hewlett-Packard Enterprise
Bloomington, MN, USA
Colin.Wahl@hpe.com

Alessandro Fanfarillo
HPC Software Solutions
Advanced Micro Devices, Inc.
Longmont, CO, USA
alessandro.fanfarillo@amd.com

***Abstract*— The Big Data Analytics Suite (BDAS) contains three classic machine learning codes: K-Means, Principal Component Analysis (PCA), and Support Vector Machine (SVM). This article describes how the 3 CPU codes, originally written in R, have been rewritten in C++ with HIP and MPI, and recast into GEMM-centric operations, taking full advantage of the heterogeneous architecture of the Frontier system. The new accelerated implementation of K-Means is now 82% GEMM-centric, PCA is 99% GEMM-centric, and finally, a new implementation in SVM will make it 20% GEMM-centric. Once completed in SVM, the entire machine learning suite will be GEMM driven. A discussion about AMD Tensile optimization of the GEMM operation adapted to extremely tall-and-skinny matrices in BDAS is included. The improvements from the original CPU R codes to the new accelerated versions, referenced to the same number of Frontier nodes in use, are 320X, 360X and 120X, respectively for K-Means, PCA, and SVM. Future integration with python and inclusion of various precision types is also briefly discussed.**

Keywords—K-Means, Principal Component Analysis, Suport Vector Machine, HIP, GEMM, AMD MI250X

I. INTRODUCTION

The Big Data Analytics Suite (BDAS) is a set of three classic machine learning algorithms [1] that were used as benchmarks for the CORAL-2 procurement[2] and that lead recently to the first "exascale" machine, *Frontier*. This original BDAS code consist of CPU-only implementation [3] of the K-Means algorithm [2], the Principal Component Analysis (PCA) method [3], and finally, the (linear) Support Vector Machine (SVM) method [4]. These three codes, originally written in the R-language, were combined with pbdMPI [5], implying that they could indeed take advantage of the parallel distributed memory architectures of the Frontier Cray-EX machine, but could not use its accelerators, or in fact any accelerator (from AMD, Nvidia, nor Intel).

The original BDAS R programs were not formulated to leverage level 2 or especially level 3 linear algebra subroutines, (BLAS 2 and BLAS 3, respectively). For maximal performance, each of the three algorithms were reformulated to leverage those libraries. The software was re-written to leverage C++ and Heterogeneous Interface for Portability (HIP) languages as well as libraries in AMD's open-source Radeon Open eCosystem Platform (ROCm™) software stack. The reformulation is mathematically equivalent to the original CPU-only benchmark but makes efficient use of the GPU resources. The optimization of all three programs is centered around General Matrix Multiply (GEMM) operations, each algorithm being rewritten to take advantage of the performance of GEMM kernels on GPUs. The GEMM dimensions in BDAS depart significantly from typical chemistry, engineering, or even synthetic benchmarks like HPL, where matrix dimensions are usually square, or at least somewhat square. In the accelerated BDAS version, matrices are extremely tall-and-skinny, with a ratio of dimensions surpassing 4 orders of magnitudes between them, as further detailed in this article. This characteristic makes the AMD Tensile optimization tool [6] crucial for obtaining good performance. We devote a section of the article to that important component of our optimization.

The primary step of "optimization" for the Frontier machine was thus to *first* rewrite the 3 BDAS codes to run relatively efficiently on AMD MI250X GPU nodes (and tested prior to that on MI60 or MI100 hardware). Further optimization of these codes was then incrementally done to get the best performance on the Cray-EX system. Note that we had considered other possibilities in this process. For example (a) the Python h2oai/h2o4gpu code[4] as suggested in the CORAL-2 document; (b) CUDA/MPI versions for K-Means and SVM that we then "hipified", plus a PCA Fortran/scaLAPACK version. However, we opted to directly translate the R codes with the goal of keeping all three implementations uniform.

In each algorithm MPI is used for the communication between nodes of the network. The specifics of the GPU-to-GPU MPI communication implemented in BDAS is described in detail in each section of the three codes.

The BDAS benchmark uses synthetic randomly generated data. However, all codes and solutions are validated using the iris data set, comparing results with the original R with pbdMPI codes.

The description in the next section is general and applies to all three BDAS codes. The three subsequent Sections will focus on the specifics of each: K-Means, PCA, and SVM. Section VI

[1] Now at Microsoft Azure, Redmond, WA, USA

[2] https://asc.llnl.gov/coral-2-benchmarks

[3] https://www.r-project.org/

[4] https://github.com/h2oai/h2o4gpu

briefly discusses the AMD Tensile optimization program used to optimize the tall-and-skinny matrix multiply operations occurring in BDAS. Section VII summarizes the performance gains from the original CPU-only code, and specific profiles for each code. We conclude in Section VIII with a short discussion on potential improvement and potential integrations with python/cython.

II. General acceleration scheme

The BDAS implementation was validated using the iris dataset that contains 150 objects (3 species), and 4 features (i.e., the sepal and petal lengths and widths). The iris dataset is a tall-and-skinny matrix (150 X 4), although tiny in size compared to the benchmark (16,000,000 X 250, discussed below). Once the run was validated, performance was tested using a large, randomly generated, dataset for the benchmark. The choice for the dimension of the benchmark is discussed in the next two paragraphs. The number of objects is generally several orders of magnitudes larger than the number of features [e.g., the few types of credit card transactions of millions of users]. This imbalance of matrix dimension has critical implications for GEMM performance and Tensile optimization, as further described below in Section VI.

The data in BDAS is distributed across a number of MPI ranks, which allows to consider any size datasets (limited in practice only by the amount of memory available on the entire system). An "ensemble" run is defined by the total amount of data to be distributed. For performance comparison purposes one ensemble is chosen to be 1.024 terabytes (TB) of distributed data (a convenient unitary unit of measurement used throughout the article). The number of input features in all benchmarks is constrained to be equal to 250, letting the number of objects to be arbitrary large.

One important common optimization scheme true to the three BDAS codes was to *increase the concentration of work* per MPI rank that is assigned to each accelerator. For instance, if one ensemble of the original R code is distributed across 2048 MPI ranks, that means that each MPI rank can hold 250,000 double precision local objects. That same ensemble uses only 32 MPI ranks in the optimized implementation, corresponding to 16,000,000 double precision *local* objects. In both cases, the R codes, and the C++/HIP codes, 1.024TB of data are used; it's the *local* matrices that have strongly different ratio of dimensions. Thus, all local matrices in the optimized codes are extremely tall-and-skinny compared to the original CPU-only R implementation; they are in fact 64 times taller and skinnier than in the original.

The initialization in BDAS is identical for all 3 algorithms. Two components need to be initialized: MPI and HIP. MPI codes require few lines of MPI initialization prior to calling any MPI function, which essentially tell the compiler to prepare for communication across the nodes in the network. On top of this, the HIP and ROCm™ libraries add a new software layer requiring initialization. The HIP initialization essentially tells the compiler that the MPI ranks are to be assigned to specific accelerators on the nodes. Because some of this is novel to many in the target audience, a complete excerpt of the MPI/HIP/ROCm™ initialization is shown next. The HIP initialization takes values from the MPI environment variable ${LOCAL_RANK}. In this example, we use the SLURM workload manager, :

```
export LOCAL_RANK=$SLURM_LOCALID
...
#include "mpi.h"
#include <hip/hip_runtime.h>
#include "hipblas.h"
#include "rocblas.h"
...
int main(int argc, const char** argv){

// MPI Initialization
MPI_Init( NULL, NULL);
int num_ranks, rank;
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &num_ranks);

// HIP + ROCm Initialization
int dev, dev_count;
char* str;
hipError_t hip_result;
if ((str = getenv("LOCAL_RANK")) != NULL) {
   hipGetDeviceCount(&dev_count);
   int local_rank = std::atoi(str);
   dev = (local_rank % dev_count);
}
hip_result = hipInit(0);
if (hip_result != hipSuccess) {return 1;}
hip_result = hipSetDevice(dev);
if (hip_result != hipSuccess) {return 1;}
hipblasHandle_t handle;
hipblasCreate(&handle);
rocblas_initialize();
...
```

This initialization is general and would apply to any code that performs a one-to-one assignment between an MPI rank and a GCD of the MI250X. The `handle` variable is used by GEMM operations and other BLAS 2 and BLAS 3 calls. The `rocblas_initialize()` function usually leads to performance improvement in the GEMM calls, and since that initialization step is done only once, prior to calling the multiple GEMM operations, it is usually beneficial. Once that initialization is completed then GPU-to-GPU MPI communication can be utilized. In practice "on device" variables (that are initialized with a `hipMalloc` allocation) can directly be used as arguments to the `MPI_*` function calls (at least on a Cray-EX machine). Relevant snippets of codes showing these features are shown in each Section of the three codes.

Combining MPI with HIP adds a new layer of potential synchronization: one between the network and the CPU, and one between the GPU and the CPU. On the one hand, an `MPI_Barrier()` function call ensures synchronization between data that is transferred from the network that is to be used onto the CPU. MPI *collectives* integrate a default barrier at the end. On the other hand, the

function `hipDeviceSynchronize()` has a similar purpose, but in this case between the GPU and the CPU. This synchronization ensures that all the hip "wavefronts" on the AMD GPU (also called "warps" on Nvidia GPU) are *all* completed at the end of the execution of the hip function. Note that `hipblasDgemm(...)` does *not* include a default synchronization at the end. The following excerpt of code (from K-Means) shows this kind of relationship between network, GPU, and CPU synchronizations:

```
...
double*       ClusterCentroids_OnDevice;
hipMalloc(
     (void**)&ClusterCentroids_OnDevice,
     NFeatures*KClusters*sizeof(realtype)
     )
...
hipblasDgemm(handle,
             HIPBLAS_OP_T,
             HIPBLAS_OP_N,
             NFeatures,
             KClusters,
             MObjects_rank,
             &alpha,
             DataMatrix_OnDevice,
             MObjects_rank,
             ClusterID_OnDevice,
             MObjects_rank,
             &beta,
             ClusterCentroids_OnDevice,
             NFeatures
             );
hipDeviceSynchronize();
MPI_Allreduce(MPI_IN_PLACE,
              &ClusterCentroids_OnDevice[0],
              NFeatures*KClusters,
              MPI_DOUBLE,
              MPI_SUM,
              MPI_COMM_WORLD);
// The MPI_Barrier() is implicit
...
```

All variables ending with "`_OnDevice`" are defined through an `hipMalloc`, which allocates memory on the device. The call to `hipblaDgemm` is concurrently executed on a GCD. The subsequent `hipDeviceSynchronize()` ensures that all wavefronts on the GCD are completed at the end of the call, and that all data are *equally* available on the GPU. Then the `MPI_Allreduce` function communicates through the network the variable `ClusterCentroids_OnDevice`. It is important to notice that that variable in the arguments of the standard MPI interface function `MPI_Allreduce` is *on the device*. In other words, no intermediary transfer from GPU to CPU is necessary! As noted above an implicit barrier resides at the end of any `MPI_Allreduce`, ensuring that once the communication is complete, all data are *equally* available on the CPU. This structure of double synchronization (network-GPU/CPU-GPU) is used in all three BDAS codes. This example of double synchronization is relatively simple. There are no general methods for dealing with potential race conditions in a more complex program [5]. However, systematically including `hipDeviceSinchronize()` and `MPI_Barrier()` calls after *all* functions, then removing them incrementally, after careful analysis of the call structure, may give a little bit of a procedural approach for keeping codes functional, and at the same time, getting them incrementally faster in the process.

Double and single precision arithmetic are both fully integrated in the optimized BDAS. Swapping from double to single precision is done through a -DDATA_BYTES variable in the Makefile, where DATA_BYTES is defined by:

```
#if (DATA_BYTES ==4)
typedef float realtype;
#else
typedef double realtype;
#endif
```

The variable `realtype` is used to define all floating-point variables, or arrays, everywhere in BDAS.

Half precision is currently under evaluation at the time of writing this article. Adding half precision implies adding the following library and `realtype` definitions:

```
#include <hip/hip_fp16.h>
#if (DATA_BYTES ==2)
   typedef __half realtype;
#if (DATA_BYTES ==22)
   typedef __half2 realtype;
```

This syntax is relatively new and is being tested using simple parallel dot product instructions on MI250X GPU, that are to eventually be integrated in BDAS. It is believed that future AMD hardware will get substantial performance benefits from the use of half precision arithmetic.

Compiling BDAS on a Cray-EX machine requires the following essential modules:

```
Either PrgEnv-cray or PrgEnv-gnu
module load rocm
module load craype-accel-amd-gfx90a
```

Once loaded, the Makefile requires using the following libraries from rocm and hipblas:

```
CXXFLAGS= -std=c++11              \
          --amdgpu-target=gfx90a \
          -O3
CXXFLAGS+= -DDATA_BYTES=${DATA_BYTES}
```

[5] https://developer.download.nvidia.com/CUDA/training/StreamsAndConcurrencyWebinar.pdf

```
CXXFLAGS+= -I${CRAY_MPICH_PREFIX}/include
CXXFLAGS+= -I${ROCM_PATH}/include
CXXFLAGS+= -fno-omit-frame-pointer      \
           -mno-omit-leaf-frame-pointer \
           -fno-optimize-sibling-calls

LDFLAGS=-L${CRAY_MPICH_PREFIX}/lib         \
        -lmpi                              \
        -L${CRAY_MPICH_BASEDIR}/../gtl/lib \
        -lmpi_gtl_hsa
LDFLAGS+=-L${ROCM_PATH}/hipblas/lib/       \
         -lhipblas                         \
         -L${ROCM_PATH}/rocblas/lib/       \
         -lrocblas \
         -L${ROCM_PATH}/lib \
         -lamdhip64 \
         -lhsa-runtime64
```

The necessary libraries in bold are specifically related to hipBLAS and rocBLAS. Hierarchically, hipBLAS is the BLAS library that marshals the AMD GPU rocBLAS library[6]. In other words, the hipBLAS library is more general and could marshal other libraries, such as cuBLAS. The other two libraries, `amdhip64` and `hsa-runtime64`, are runtime hip and rocr libraries. The variable `${ROCM_PATH}` is defined through `module load rocm`. The remaining libraries are typical MPI libraries on Cray-EX.

The next three Sections describe specifically how optimization was performed for each of the BDAS codes: K-Means, PCA, and SVM, in alphabetic order.

III. ACCELERATING K-MEANS

The optimized K-Means code was based on the original R algorithm, where each essential part of the serial R code has been extracted and re-written in C++ with calls to `hipBLAS` libraries. Subsequently, the code was parallelized with MPI. One conceptual difference between the original R with pbdMPI code and the new C++/HIP with MPI is in the sequence of work: in the original R code only one cluster size is treated. For example, for a given dataset if K=2, then K=3, and finally K=4 are evaluated, then the R code would solve them sequentially. In the optimized version those three cluster sizes are *stacked* in parallel, which translates to "K" being of dimension KClusters = 2+3+4 = 9. This single dimension is important for defining the GEMM operations and its optimization. Note that since K-Means is an "NP-hard" problem this scheme of launching multiple cluster dimensions (2, 3, and 4) at the same time can easily be expanded to launching multiple initial conditions at the same time (an implication of NP-hardness) for a given cluster dimension, K.

The reformulated K-Means Lloyd algorithm is computing

$$\arg\min_{S} \sum_{i=1}^{K} \sum_{x \in S_i} ||x - \mu_i||^2,$$

where the summation index, S_i, is the set of data points in the i-th cluster, K is the number of clusters, with multiple problems that can be stacked, or not, and μ_i is the centroid of the i-th cluster. The solution of this optimization problem is obtained using an algorithm based on alternating between updating the clusters and updating the centroids until a fixed-point solution is reached, as further shown in a code snippet below.

The algorithm proceeds in two steps. First, the set of centroids is assumed fixed (and randomly initialized if it is the first iteration). With this assumption the algorithm computes

$$\|x - \mu_i\|^2 = x^T x - 2x^T \mu_i + \mu_i^T \mu_i, \quad (1)$$

for each combination of data point and cluster, assigning each data point to the cluster that minimizes the distance between the point and the cluster centroid. Each data point is assigned to multiple clusters if multiple K-Means problems are stacked together. Next, the data points are assumed to belong to fixed clusters, and the centroids of these new clusters are computed. This cluster centroid equals

$$\mu_i = \frac{1}{|S_i|} \sum_{x \in S_i} x,$$

for the i-th cluster. Then this sequence of operations is repeated until all cluster memberships no longer vary, thus attaining a fixed-point solution.

The K-Means program is relatively small: 1360 lines of C++/HIP/MPI coding. The program starts by computing the dot product represented by the first element of Eq. (1), on the device and in parallel. The next paragraph shows how tiling is done for that dot product instruction. Note that this dot product code is being used for evaluating half precision. That structure is relatively general, and it occurs also in SVM. It should easily be extended to other programs. The dot product method is called (in the main) using the following structure:

```
...
const int TileSize=256;
const int TileCount=256;
long m_r =MObjects_rank%(TileCount*TileSize);
long m_d =MObjects_rank - m_r;
...
hipLaunchKernelGGL((dot_products),
                   0,0, m_r, m_d,
                   MObjects_rank,
                   NFeatures,
                   DataMatrix_OnDevice,
                   DataMatrixSquared_OnDevice
                   );
```

[6] https://rocblas.readthedocs.io/en/latest/

This dot_products (notice the plural) function is tiling the threads (generating "wavefronts" of computations on the GPU, in waves of chosen dimensions 256 X 256 tiles). This dot_products function has the following syntax:

```
__global__ void dot_products(
   long m_r, long m_d,
   long M  , long N,
   realtype * a_d,
   realtype * norm_d)
{
 long blockID  =
 hipBlockIdx_x +
 hipBlockIdx_y * hipGridDim_x +
 hipBlockIdx_z * hipGridDim_x * hipGridDim_y;

 long threadID =
 hipThreadIdx_z*(
 hipBlockDim_x * hipBlockDim_y ) +
 hipThreadIdx_y *hipBlockDim_x +
 hipThreadIdx_x;

 long tid    = threadID;
 long i1     = blockID*(TileSize) + tid;
 long stride = TileSize * 2 * TileCount;
 long i      = blockID*(TileSize*2) + tid;

 if (i1 < m_r + m_d)
    dot_product(i1, M, N, a_d, norm_d);
}
```

For each `threadID`, the `dot_product` (notice the singular) function calculated on the device is called. This local `dot_product` function takes the form of a usual dot product:

```
__device__ void dot_product(
                  long tid,
                  long M,
                  long N,
                  realtype * a_d,
                  realtype* norm_d)
{
   realtype sum = 0.0;
   for(long k1 = 0; k1 < N; k1++) {
      sum += a_d[tid + k1*M]*a_d[tid + k1*M];
   }
   norm_d[tid] = sum;

}
```

Next, recall that the K-Means algorithm is a fixed-point iteration. The pseudocode takes the following form, in terms of HIP and MPI function calls. Most variables are on device, except for the reduction step:

```
for (int outer=0; outer < FIXEDPOINT; outer++)
{
 (1)hipblasDgemm
  in:  DataMatrix_OnDevice,
  in:  ClusterCentroids_OnDevice,
  out: DataXCentroidMatrix_OnDevice
 (2)hipblasDdot
  in:  ClusterCentroids_OnDevice
  out: CentroidMatrixSquared_OnDevice
 (3)hipLaunchKernelGGL(assign_cluster)
  in: DataXCentroidMatrix_OnDevice
  in: DataMatrixSquared_OnDevice
  out: ClusterID_OnDevice
 (4)reduction tiled;
  in: ClusterID_OnDevice
  out: NumOfPointsPerCluster_OnHost
 (5)MPI_Allreduce
  inout: &NumOfPointsPerCluster_OnHost

 (6)hipMemcpy
  out: NumOfPointsPerCluster_OnDevice)
 (7)hipLaunchKernelGGL (normalize_weights)
  in: NumOfPointsPerCluster_OnDevice
  out: ClusterID_OnDevice

 (8)hipblasDgemm
  in:  DataMatrix_OnDevice,
  in:  ClusterID_OnDevice,
  out: ClusterCentroids_OnDevice
 (9)MPI_Allreduce
  inout: &ClusterCentroids_OnDevice
}
```

Line (2) of the above pseudocode corresponds to the last term of Eq. (1). Line (3) assigns each data point to one cluster, or multiple clusters if multiple K-Means algorithms are being executed at once, i.e., Clusters=2+3+4=9. This kernel also populates the important `ClusterID_OnDevice` matrix, referred to as K. That matrix holds one row for each data point and one column for each cluster, where the (i,j) entry will be set to *one* if the i-th data point is in cluster j and *zero* otherwise, thus defining their membership. It involves the 2nd DGEMM. The reduction step at line (4) counts the number of points in each cluster and then the normalize_weights kernel divides each 1.0 value in the K matrix by the number of points in the appropriate cluster to normalize the values. The 2nd hipBLAS DGEMM operation at line (8) is used to compute the new cluster centroids.

Structurally, Lines (1) through (5) constitute one balancing swing of the fixed-point iteration, which corresponds to assigning the clusters. Then lines (6) through (9) do the counter-balancing swing of that same fixed-point iteration, which corresponds to finding the new centroids. The process continues until the two counter-balancing sections of code stop changing, thus reaching a fixed-point. In practice, that implies that the pseudocode translates to having two pairs of `hipblasDgemm + MPI_Allreduce` calls [one at lines (1) and (5), and another pair at lines (8) and (9)]. Note that the first `MPI_Allreduce` is using variables that are on host, while the second is doing GPU-to-GPU communication (as discussed previously in Section II). The next paragraph further explores the specific characteristics of those two GEMM operations using the benchmark dimensions of a typical BDAS case.

In general, a GEMM operation performs a matrix-matrix multiply between a first matrix A of dimension M times k (not to be confused with the K clusters in K-means), multiplied by a second matrix B of dimensions k times N, resulting in a matrix C of dimensions corresponding to the

outer dimensions of the previous two: an M times N matrix. In K-Means, the dimensions of the GEMM (and the corresponding MPI_Allreduce) operations are respectively:

1st GEMM (none transposed):

```
M = MObjects_rank = 16,000,000
N = KClusters     = 9
k = NFeatures     = 250
```

1st MPI_Allreduce: `KClusters = 9`

2nd GEMM (first transposed):

```
M = NFeatures     = 250
N = KClusters     = 9
k = MObjects_rank = 16,000,000
```

2nd MPI_Allreduce: `NFeatures*KClusters = 2250.`

The dimensions of the benchmark, used for analyzing the performance of the algorithm, are also shown. The software Tensile, which is discussed in Section VI, optimizes the implementation of these DGEMM operations for the MI250X. Profiling runs are discussed later in Section VII.A.

IV. Accelerating PCA

In general, there are two approaches for evaluating principal components from a dataset: (1) through a calculation of the singular value decomposition of the non-symmetric matrix of dimensions `MObjects` times `NFeatures`, or (2) through a calculation of the eigenvalues of the symmetric variance matrix of dimensions `NFeatures` times `NFeatures` [7]. In practice, the second method is more amenable to execution on the GPU, because computing the variance (or covariance) utilizes a GEMM operation. While there are probably SVD codes that are optimized to take advantage of GEMM operations when constructing the fundamental subspaces ([7], chapter 6), we have not considered that route in this implementation. Essentially, we followed the same route as in the original R code: computing the eigenvalues of the variance matrix.

PCA (compared to the other two BDAS codes) is very simple. Because of this, the pseudocode is essentially the C++/HIP code itself. Once all variables are allocated on device (through hipMalloc), the actual algorithm reads as:

```
...
if (rank == 0) {
   eig.init_syevd(
      Variance_OnHost,
      NFeatures);
}
hipblasDgemm(handle,
             HIPBLAS_OP_T,
             HIPBLAS_OP_N,
             NFeatures,
             NFeatures,
             MObjects_rank,
             &alpha,
             DataMatrix_OnDevice,
             MObjects_rank,
             DataMatrix_OnDevice,
             MObjects_rank,
             &beta,
             Variance_OnDevice,
             NFeatures
             );
MPI_Reduce(&Variance_OnDevice[0],
           &Variance_OnHost[0],
           NFeatures*NFeatures,
           MPI_DOUBLE,
           MPI_SUM,
           0,
           MPI_COMM_WORLD
          );
if (rank == 0) {
   eig.compute_eigenvalues(Variance_OnHost);
}
```

Both functions `init_sysevd` and `compute_eigenvalues` contain respectively the usual LAPACK initialization and execution for the function `dsyevd` from NETLIB (or cray-libsci), on host:

```
dsyevd(&jobz,
       &uplo,
       &matrix_size,
       matrix,
       &matrix_size,
       eigenvalues.get(),
       &wkopt,
       &lwork,
       &iwkopt,
       &liwork,
       &info);
...
```

The DGEMM dimensions are:

```
M = NFeatures     = 250
N = NFeatures     = 250
K = MObjects_rank = 16,000,000,
```

where the matrix-matrix product is applied to itself, as $A^T A$. The LAPACK routine "DSYRK" also implements this type of operation (as it is done in the original R code). However, that routine was not optimized on GPU through AMD Tensile. It is thus important to specifically *choose* DGEMM even though the matrix product is technically on itself, that is, a DSYRK.

The above code is our final implementation, which turns out to be the simplest approach among three that we had considered. A second approach was a FORTRAN implementation that called two parallel scaLAPACK functions: PDGEMM, followed by the parallel eigenvalue solver, PDSYEVD. That method has the disadvantage to not take directly advantage of the AMD Tensile optimizer. Moreover, that formulation creates excessive communication within the tiny eigenvalue solver (since the variance is only 250 X 250 in size). A third approach that we considered was to manually subdivide the tall-and-skinny matrix into a sum of "squarer" matrices, trying to bring the extremely tall-and-skinny matrix dimensions into a sum of matrices that have dimensions that are closer to those seen in typical quantum chemistry, or engineering applications, then multiplying the local matrices together through a series of

matrix-like dot products. However, after several tests, we realized that this option was essentially one subset of the parameter-space that the AMD Tensile optimizer would eventually explore. Performance was no better, and coding was cumbersome. We find that it is best to let the AMD Tensile optimizer find the best subdivision, rather than to try to figure one out manually. The best option we found is the first approach shown above: to execute DGEMM independently on each GPU, take advantage of the AMD tensile optimization, send the result to rank 0, where finally the tiny eigenvalue problem is solved on host.

Notice an interesting twist of *combined*-MPI/HIP programming, where the `MPI_Reduce` is communicating from all ranks to rank 0 and transferring from the device to the host, all at the same time! In other words, it is communicating the *on-device* variance from all ranks, onto the *on-host* variable at rank 0, only through the list of MPI arguments (shown in bold in the above excerpt). The *on-host* variable is then ready for the next step: the eigenvalue solver that is executed *on-host*.

V. ACCELERATING SVM

Due to the sequential nature of the Nelder-Mead algorithm, accelerating SVM represents a bigger challenge than the other two BDAS codes. This Section starts by describing the Nelder-Mead algorithm. Then we describe the implementation of the cost function using a matrix-vector operation. We finally insert the main components necessary for obtaining an efficient GPU optimized algorithm, ending the Section by showing how a GEMM operation can be integrated in a recent implementation.

The Nelder-Mead implementation is directly based on the original Pascal program written by J.C. Nash [8]. Note that the original BDAS R code also uses that same algorithm. The principle of the Nelder-Mead algorithm is to evaluate a cost function "at each point (vertex) of the simplex [the structure formed by (n+1) points] and the vertex having the highest function value is replaced by a new point with a lower function value." Nash continues by stating the "four main operations which are made on the simplex: reflection, expansion, reduction, and contraction" ([8], page168). The actual Pascal implementation starts at page 173 of Nash's book. We first translated Nash's Pascal code into C++ with HIP (without MPI). After testing the single node program's performance, we subsequently created a multi-node implementation using MPI.

The Nelder-Mead algorithm includes 5 calls to a cost function, corresponding to the 4 main operations described above, plus one initialization. The pseudocode centered around those cost function is as follows:

```
Initialize: 1st cost_function
Loop over 2nd cost_function
if (Highest > Lowest && Lowest > tolerance)
     3rd cost_function
     if (Reflection < Lowest)
          4th cost_function
     else (Reflection >= Lowest)
               5th cost_function
endif
```

Note that the 4th and 5th calls are exclusive, constituting an either-or branch. The sequential nature of the algorithm stands out from the if-then-else conditions across the four main operations on the simplex. Notice that the cost function loops over the series of saved values of the polytopes. We first show in the next paragraph that the cost function includes a matrix-vector operations: DGEMv.

The SVM algorithm's cost function involves a *hinge-loss* function of the form $max(0, 1 - y_i\ (w^T x))$, where $w^T x$ is the linear intended output, and y_i, the classifier score, that is equal to ±1, used to separate the data into two ensembles (i.e., the "support vector" that defines the hyperplane that is separating the two ensembles). In SVM, we wish to minimize (using the Nelder-Mead algorithm) the following cost function:

$$\frac{1}{n}\sum_{i=1}^{n}\max\left(0,1-y_i(\boldsymbol{w}^T\boldsymbol{x})\right)+\lambda||\boldsymbol{w}||^2,$$

where $\lambda = \frac{1}{2}n$. The computationally intensive part of the algorithm is the matrix-vector operation, $w^T x$. A reduction calculates the resulting hinge-loss function. The following snippet shows the hinge loss function with the significant components of the functions call:

```
double hinge_loss(double * Weights,
                  int hipDeviceRank)
{
 double alpha = 1.0, beta = 0.0;


 hipblasDgemv(handle_global,
              HIPBLAS_OP_N,
              MObjects,
              NFeatures,
              &alpha,
              DataMatrix_Device,
              MObjects,
              Weights,
              1,
              &beta,
              ComputedSpecies_Device,
              1
              );

   // Reduction outputs hinge-loss cost.
   double out =
   reduction(ComputedSpecies_Device,
             Species_Device,
             MObjects,
             hipDeviceRank);

   // Sum reduces the cost across ranks.
   MPI_Request request;
   MPI_Iallreduce(MPI_IN_PLACE,
                  &out,
                  1,
```

```
                    MPI_DOUBLE,
                    MPI_SUM,
                    MPI_COMM_WORLD,
                    &request);

   double norm = 0.0;
   hipblasDnrm2(handle,
                NFeatures,
                Weights,
                1,
                &norm);
   MPI_Wait(&request, MPI_STATUS_IGNORE);
   // L2 Regularization
   out = 1.0/regularizationParameter*
  (out + 0.5*norm);

 return out;
}
```

This hinge loss function is the `cost_function` used in the Nelder-Mead algorithm shown above. The `hipblasDgemv` operation is part of the classifier score computation and is the most time-consuming part of SVM.

The snippet of code shows that MPI communication uses only one non-blocking reduction and is extremely minimal in SVM, even when comparing with the other two BDAS programs. The reason for having an MPI reduction is the hinge loss function output that is *locally* reduced on GPU, and that needs to be reduced *globally* across the MPI ranks. The regularization parameter is chosen to be identical to the one defined in the original R code. This version of SVM shown above does not have any GEMM operation.

A second version of SVM, which is currently under development (Makefile with the value --DGEMM=1), incorporates GEMM into SVM. When introducing the Nelder-Mead algorithm, above, it was stated that the *second* call to the hinge loss function is within a loop. Nash's literal transcription of the Nelder-Mead Pascal code [8] reads as:

```
...
-- STEP 10
if calcvert then
     begin
     for j := 1 to nl do
        begin
        if j<>L then
        begin
        for i := 1 to n do Bvec[i] := P[i,j];
        f:= fininfn(n,Bvec,Workdata,notcomp);
        ...
```

In BDAS, the function `fininfn` contains an `hipblasDgemv`. As the Pascal code above shows, that matrix-vector operation exists within a loop: a loop over a set of matrix-vectors is essentially a matrix-matrix operation, i.e., a single `hipblasDgemm`. In practice, that implies having to insert the GEMM operation directly inside the Nelder-Mead algorithm (which makes the new Nelder-Mead implementation "unextractable" from SVM, so it cannot be used elsewhere with, say, a quadratic loss function). The new `NelderMead_GEMM` code now contains the following call to:

```
hipblasDgemm(handle,
             HIPBLAS_OP_N,
             HIPBLAS_OP_N,
             MObjects_rank,
             NFeatures,
             NFeatures,
             &alf,
             DataMatrix_Device,
             MObjects_rank,
             WeightsMatrix_Device,
             NFeatures,
             &bet,
             SpeciesMatrix_Device,
             MObjects_rank);
```

The DGEMM dimensions in SVM are:

```
M = MObjects_rank = 16,000,000
N = NFeatures     = 250
K = NFeatures     = 250
```

These dimensions correspond to a product between a tall-and-skinny matrix and a tiny 250X250 matrix, a quite different aspect ratio from the ones in K-Means or PCA.

In general, replacing a sequence of matrix-vectors (computed within a loop) with one single matrix-matrix operation essentially corresponds to transferring a problem of larger runtime into a problem of larger memory (i.e., trading time, for space). At the time of writing this paper, we are considering batching the DGEMM into two calls (reducing k to 125), because the above approach surpasses the amount of local memory available (for those benchmark dimensions).

VI. TENSILE OPTIMIZATION

Currently, every time a GEMM API is invoked in rocBLAS, the actual computation is performed by a separate and specialized library called Tensile.

Tensile [6] is an open source [7] tool able to auto generate highly performing GPU assembly kernels optimized for specific GEMM sizes. Due to the complexity of modern GPUs not all problem sizes can be efficiently handled in the same way. Some implementations for a specific problem size will perform differently from others based on many factors, such as: number of workgroups, cache accesses, type of MFMA instructions used and others. The overall idea is to have a fixed set of sized-tuned assembly kernels generated offline after a thorough tuning process and ship them in an official ROCm™ release.

Although this approach is extremely effective when the GEMM sizes submitted to rocBLAS match exactly the sizes used during tuning, the performance for GEMM sizes not considered during tuning can vary. This variation can happen unpredictably between ROCm releases due to newly added size-tuned kernels in Tensile.

[7] https://github.com/ROCmSoftwarePlatform/Tensile

Furthermore, the specific sizes used by BDAS are particularly challenging to handle due to large imbalances between the number of rows of the first and second matrix (16Mx250, tall and skinny).

For this reason, we decided to create a bespoke Tensile library specifically tuned for all the GEMM sizes invoked during the execution of the three test cases, and only for those sizes.

AMD generated a special library for the BDAS application. In practice, a custom Tensile generated library can be loaded and used on top of a given ROCm library. Currently, the dimensions used in BDAS (16,000,000 X 250 matrices) are part of rocm/5.4.3.

VII. Performance profiles

This Section combines the performances of the three BDAS codes, focusing on their GEMM and GEMv profiles. The method for validating each algorithm is also discussed at the beginning of each subsection. All algorithms are validated using the iris dataset (three species of Iris setosa, Iris virginica, and Iris versicolor). There are 150 objects and 4 features (the sepal and petal lengths and widths) in that dataset. The solution is known and thus the quality of each algorithm can be evaluated.

It is important to realize that the runtime shown below, for K-Means and PCA, are fast only thanks to the Tensile optimization. Any ROCm *non-optimized* library would give runtimes much slower, slower perhaps by an order of magnitude (~10 to 20X slower). *The Tensile optimization is essential in obtaining these fast runtimes.*

A. K-Means

The K-Means implementation is validated by varying the initial conditions of the clusters (K-Means being an "NP-hard" problem). The K-Means algorithm is tested by setting K=3, in parallel (e.g., 2 MPI ranks). The Rand index function [9] defines the "distance" between two ensembles, a characterization of the overlap between the known solution (the 3 species) and the K-Means solution (the 3 clusters). Depending on the initial condition, the Rand index distance varies between 71.48% and the maximum obtained: 90.55%, an excellent overlap between the simulation and the solution.

The benchmark runtime for the 16,000,000 X 250 local random matrix, using 32 MPI ranks on 4 nodes, is 0.23 seconds. The original R CPU-only code requires 2048 ranks distributed across 32 nodes, with local matrices of 250,000 X 250, to simulate that same terabyte of data. That means that the optimized code requires 8 times fewer nodes. The runtime from the CPU-only code is 9.2 seconds for that same 1.024TB ensemble. That means that the optimized code is 40 times faster. Multiplying those two factors (nodes required and runtimes) the new accelerated K-Means version is thus 320X better, i.e., the same work can be done on 8 times fewer nodes, in 40 times faster runtime.

Table I shows the profile using `$ROCM_PATH/bin/rocprof` preloaded to the usual SLURM run script, for the 1.024TB ensemble, running on the MI250X nodes of a system equivalent to the Frontier machine (the machine "crusher").

Table I. K-Means rocprof profile

Method	Percentage	DGEMM
Cijk_Alik_Bljk (32x16x32)	51.6	**82.2**
Cijk_Ailk_Bljk (1024x32x8)	30.6	
dot_product	12.7	
assign_cluster	2.3	
normalize_weights	1.8	
reduction_hip_tiled	0.9	
rocblas_dot_kernel_magsq	0.04	
Cijk_D	0.004	

It shows that the optimized K-Means code has been rewritten so that the algorithm depends mostly (82%) on the GEMM calls, which have been optimized by Tensile as described in Section VI for these two extremely tall-and-skinny matrices. The MPI runtime spent in the two `MPI_Allreduce` communication across only 4 nodes is negligeable (fraction of a percent). We also tested weak scaling of the code, by increasing the size of an ensemble up to 128TB per ensemble. We find that MPI communication becomes noticeable (>2%) beyond 32TB, or 256 MPI ranks. In general, we suggest splitting the problem into 1TB problems, when possible, to avoid potential load imbalance. Load imbalance depends on the dataset. Realistic datasets may show larger imbalance than the randomly generated datapoints we use for the benchmark. From this profile K-Means is more than 80% compute bound and dominated by GEMM operations. The 4 functions in bold, in Table I are rocm libraries, while the other four are K-Means function that were described in Section III.

B. PCA

PCA is first validated by comparing the eigenvalues of the centered and normalized iris dataset from the R code. The R code gives eigenvalues:

```
[1]2.91849782 0.91403047 0.14675688 0.02071484
```

The output of the optimized PCA eigenvalues gives identical values:

```
Eigenvalues =
            2.071483642862e-02
            1.467568755713e-01
            9.140304714681e-01
            2.918497816532e+00
```

The benchmark runtime for the 16,000,000 X 250 local matrix, using 32 MPI ranks, is 0.15 seconds. Equivalently, the original R code takes 6.8 seconds to complete the 1.024TB ensemble run (using 2048 MPI ranks, as in K-Means). The optimized code is 45 times faster. On a per ensemble basis the optimized code is 45X faster, and

on a per node basis, as is calculated in K-Means, it is 360X faster, i.e., the same work can be done on 8 times fewer nodes, in 45 times faster runtime.

Table II. PCA rocprof profile

Method	Percentage	DGEMM
Cijk_Alik_Bljk (128x256x16)	99.992	**99.992**

The rocprof profile for PCA is shown in Table II. It shows that essentially all (99.992%) of the runtime is spent in the matrix-matrix multiply operation, optimized for extremely tall-and-skinny matrices using AMD Tensile.

The `MPI_Reduce` runtime for transferring to rank 0, as well as the runtime for calculating the tiny 250X250 variance matrix eigenvalues (through `DSYEVD`), are both negligeable. PCA is essentially 100% compute bound.

C. SVM

The optimized implementation of SVM is validated by separating one iris species from the other two and letting SVM find the support vector that separates those two sets. The accuracy of that separation is computed to be 100%.

The benchmark time for SVM on Frontier using 4 nodes (1 ensemble of 1.024 TB) is 13.47 seconds. The R CPU-code does the same work in 202 seconds. That is, 15X faster on GPU. Since the same work can be done on 8 times fewer nodes, as for the other two codes, that corresponds to a 120X improvement. The performance gain in SVM is not as large as in K-Means and PCA, but it is expected that this gap will close with the new version that is rewritten to use GEMM operations.

Table III SVM rocprof profile

Method	Percentage	DGEMM
		0
gemvn_kernel	**99.16**	
reduction_hip_tiled	0.74	
temp_accumulate	0.07	
rocblas_reduction_strided	0.015	

The SVM profile shown in Table III shows that most of the time is spent in the DGEMv call that occurs inside the hinge-loss function. DGEMv cannot be optimized with AMD Tensile (since the varied parameter from the parameter space search algorithm, i.e., various strides in a matrix are not available). SVM is shown to be essentially 100% memory bound.

Recall that in Section V above we have shown that the Nelder-Mead algorithm contains 5 calls to the hinge loss function, and that the second call (STEP 10 in Nash's algorithm) includes an inner loop call to DGEMv that we transformed into a DGEMM.

Table IV. Nelder-Mead cost function calls for the tiny iris and the large random benchmark datasets

Cost function call	Iris	Random
1st (initialization)	1	1
2nd (DGEMM loop)	**5**	**250**
3rd Reflection	248	125
4th lower	75	0
5th upper	173	125

Table IV shows the number of calls inside Nelder-Mead, for both the Iris dataset and for the large synthetic random dataset. The proportions of DGEMM (2nd call) are inverted between the two datasets: The iris dataset has few DGEMM calls compared to the other 4 DGEMv calls, while the randomly generated large dataset has about half of the DGEMM calls, and the other half being DGEMv. We believe that larger realistic datasets would likely have between be 20%-25% of the total calls in DGEMM. Performance analysis with this new implementation is under development at the time of writing this article.

VIII. Conclusion

This article describes a C++ with HIP and MPI implementation of the classic K-Means, PCA, and linear SVM machine learning algorithms. We show that leveraging BLAS 3 GEMM operations in these three programs can significantly improve their performance, by taking advantage of the AMD Tensile GEMM optimizer. We find that K-Means, PCA, and SVM are respectively 320X, 360X, and 120X faster than the original CPU-only implementations. The K-Means code spends 82% of its time in GEMM operations, for PCA this figure is 99%, and we have shown that SVM contains a potential for the runtime to spend over 20% of its time in GEMM operations in the second call to the hinge loss function inside Nelder-Mead (still under investigation).

These techniques could easily be integrated into other modern statistical learning algorithms, such as K-Means++ [10], SVD [11], or non-linear SVM [1].

Python being one of the preferred languages of machine learning, we believe that these techniques written in C++ with HIP and MPI could eventually be included in a python library or included and integrated using cython with HIP. Dragon or mpi4py could also be considered to replace the MPI instructions, since the structure of internode communication has been kept minimalistic. HIP-Python (like the existing CUDA-Python) is under development. Such integration could also be considered, soon.

This accelerated BDAS code was created as part of the CORAL-2 procurement. The optimized BDAS code has been provided to Oak Ridge National Laboratory.

Acknowledgment

We thank Steven Abbott, John Baron, Joe Glenski, Faisal Hadi, Nick Hill, Krishna Kandalla, Timothy Mattox, Luke Rosko, Kevin Thomas, Norm Troullier, Trey White, and Xin Ye at HPE, the system administrators at HPE and ORNL, as well as Paul Bauman, Ian Bogle and Nicholas Malaya at AMD for their help and discussions on various possible experiments which lead to our optimization.

REFERENCES

[1] Hastie, T., Tibshirana, R., Friedman, J., *The Elements of Statistical Learning* (Springer, New York, 2012).

[2] Lloyd, S.P., *Least square quantization in PCM*, IEEE Transactions on Information Theory **28** (129-137), March 1982.

[3] Pearson K. F.R.S., LIII. On lines and planes of closest fit to systems of points in space, The London, Edinburgh, and Dublin Philosophical Magasine and Journal of Science, **2**(11), pp. 559-572, 1901.

[4] Cortes, C. and Vapnik V., *Support-vector networks*, Machine Learning **20**, 273-297 (1995).

[5] Ostrouchov, G., Chen, W.-C., Schmidt, D., Patel, P., *Programming with Big Data in R*, (Oak Ridge National Laboratory and University of Tennessee, 2012).

[6] D. E. Tanner, "Tensile: Auto-Tuning GEMM GPU Assembly for All Problem Sizes," 2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), Vancouver, BC, Canada, 2018, pp. 1066-1075, doi: 10.1109/IPDPSW.2018.00165.

[7] Elden, L., *Matrix Methods in Data Mining and Pattern Recognition*, (SIAM, Philadelphia, 2007).

[8] Nash, J.C., *Compact Numerical Methods for Computers, Linear Algebra and Function minimisation* (Adam Hilger, New York, 1979).

[9] Rand, W. M., *Objective Criteria for the Evaluation of Clustering Methods*, J. American Statistical Association, **66**, No. 336, pp. 846-850 (1971).

[10] Vassilvitskii, A.D., *K-Means++: the Advantages of Careful Seeding*, Proc. of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SIAM, Philadelphia, pp. 1027-1035.

[11] Berry, M.W., *Large-Scale Sparse Singular Value Computations*, Int. J. High Performance Computing Applications, **6**, Issue 1, pp. 13-49 (1992).