ExternalBuffer in Matrix class

There are at least 4 different implementations of the Matrix class that have over time diverged in their implementation in respect to how the external buffer case is handled. The external buffer case is when the matrix class does not actually own it's own memory and is pointing to an external buffer that is managed separately. A deviceID of MANAGEDEXTERN used to be the way this was done, however we have now moved to setting a flag m\_externalBuffer in the common header to signify an eternal buffer. We have two instances of this in our code today:

1. Column Slices were implemented using this feature. The idea is you only want to reference a portion of a full matrix, but don't want to copy the contents to a new matrix for efficiency reasons. In this case the slice can be modified just like a real matrix, and it is the programmers responsibility to ensure that the lifetime of the underlying matrix is longer than any of it's slices. NOTE: lifetime management is not taken care of for you, so be careful
2. PTask Buffers - PTask is our solution for using multiple GPUs. It uses a filter graph based approach for accelerating GPU applications. PTask executes a graph and calls each of it's tasks as the inputs are available. In CNTK most of these inputs are numeric arrays with an associated Matrix header metadata. We wrap the buffer in a Matrix shell with external buffers set, and call the normal processing methods.

Both of these uses are similar, but slightly different as well. We believe that we can use the same implementations to satisfy both sets of needs. So here are the definitions:

Matrix(const size\_t numRows, const size\_t numCols, ElemType \*pArray, const size\_t matrixFlags=matrixFlagNormal, short deviceId=AUTOPLACEMATRIX, const size\_t nnz=0);

* Matrix constructor that constructs a matrix from a buffer pointer and some flags. The behavior depends on the flags. In all cases dimensions, format (from the matrixFlags), deviceId and nnz (for sparse representations) are copied:
  + matrixFlagDontOwnBuffer - in this case the pArray pointer is set as the m\_pArray of the matrix and m\_externalBuffer = true
  + matrixFlagSetValueOnDevice - if set this signifies that the buffer is on the proper device, but needs to be copied to newly allocated space for the m\_pArray, m\_externalBuffer = false
  + neither set - the buffer is on the CPU and device memory is allocated and then the buffer is copied over, m\_externalBuffer = false

Matrix(const Matrix<ElemType>& deepCopyFrom, short deviceId=AUTOPLACEMATRIX); //copy constructor, deep copy

* Matrix constructor that constructs a matrix from an existing matrix, Dimensions, format, and other elements are also copied:
  + deepCopyFrom - regardless of if m\_externalBuffer is set or not, a new buffer is allocated and the contents of the deepCopyFrom are copied to the new buffer. m\_externalBuffer = false;
  + NOTE: use move constructor or SetValue with matrixFlagDontOwnBuffer if an externalBuffer at the destination is desired

Matrix<ElemType>& operator=(const Matrix<ElemType>& deepCopyFrom); //assignment operator, deep copy

* assignment operator copies from one matrix to another. In all cases , dimensions, format, and other members are copied, m\_externalBuffer is left unchanged, and copy of the buffer is buffer content only:
  + destination normal, deepCopyFrom is external - destination is resized as necessary and then copy.
  + destination is external, deepCopyFrom can be either - If the destination would require a resize, an exception is thrown, otherwise copy.

Matrix(Matrix<ElemType>&& moveFrom); //move constructor, shallow copy

* constructor with move semantics copies from one matrix to another:
  + moveFrom is bitwise copied to the newly created matrix. So it is an exact copy of previous matrix (which is going to be discarded without destructors running)

Matrix<ElemType>& operator=(Matrix<ElemType>&& moveFrom); //move operator, shallow copy

* assignment operator with move semantics copies from one matrix to another:
  + destination normal - In this case existing buffers are freed, and then everything is bitwise copied (including m\_externalBuffer flag).
  + destination is external - bitwise copy over everything (including m\_externalBuffer flag)

void SetValue(const Matrix<ElemType>& deepCopyFrom);

* Straight copy from one buffer to another, irrespective of m\_external flags, which remain unchanged. If the destination is not large enough, it will be resized. If buffer mismatch occurs and the destination is m\_externalBuffer, it will throw an exception.

void SetValue(const size\_t numRows, const size\_t numCols, ElemType \*pArray, const size\_t matrixFlags=matrixFlagNormal, int deviceId=MANAGEDEXTERN);

* SetValue with a buffer pointer copies the contents of that buffer to the matrix, resizing the destination as necessary. Also sets the format (through a mask of the matrixFlags) and deviceId of the matrix:
  + matrixFlagDontOwnBuffer set, destination normal - Free the contents of the current array buffer, replace pointer, dimensions, m\_externalBuffer = true
  + matrixFlagDontOwnBuffer set, destination external - replace pointer and dimensions, m\_externalBuffer = true
  + matrixFlagSetValueOnDevice set, destination normal - the buffer is on the proper device, resize destination as necessary, set the dimensions and copy buffer to the current array, m\_externalBuffer = false
  + matrixFlagSetValueOnDevice set, destination external - the buffer is on the proper device, throw if dimensions are incompatible, set the dimensions and copy buffer content to the current array location, m\_externalBuffer = false
  + no flags set, destination normal - the buffer is on the CPU, resize destination as necessary, set the dimensions and copy buffer to the current array, m\_externalBuffer = false
  + no flags set, destination external - the buffer is on the CPU, throw if dimensions are incompatible, set the dimensions and copy buffer content to the current array location, m\_externalBuffer = false