Specification for mathematics library with architecture-specific interface. Version 2.00

Contents

1. InitLibrary. Library Initialization.

- Library initialization. This function must be called one time when library loaded, before use library.
- Initialize internal variables of library.

In the current version of library, variables, requires initialization is absent. Function reserved for add new functionality.

Input parameters: none

Output parameters:

Status:
Reserved for extend library functionality.

*long uint Status (__stdcall *InitLibrary) ( );*

Notes for this document:

For unsigned numbers *long uint* width is 32 bits, *long long uint* - 64 bits.

For signed numbers *long int* width is 32 bits, *long long int* - 64 bits.

```
// Initialization
FUNCTION InitLibrary()
      : Uint32;  // Status return
```

2. Group of functions for matrix multiplication by Base Cycle Version 1 (for big matrices, classic schematics).

- Main Calculation Procedure (MCP) of operation «Matrix Multiplication» → multiply one A-macro column by sequence of B-matrix blocks (illustrated in pictures Mutr_Mult_a.GIF, Mutr_Mult_b.gif).
- All pointers must be 64-bit, aligned by 32 bytes (5 low bits of pointers must be zero).
- Length of line (Hb) selected by select one of function in this group, by counter of unrolled cycle and additional addend. See formulas in the functions description.
- Number of lines in the B matrix block (Wb) selected by select one of function in this group. It not transferred as explicit parameter. See decoding and description of functions names.
- All counters is 32-bit integer values.
- A-atom contains 2 quads of numbers (64 bytes).
- B-atom contains 4 quads of numbers (128 bytes).

- Store of result matrix C executed as add to memory (add with current elements values, not write-only to memory). This means C matrix must be cleared before operation if addition to current values not required.
- When prepare input parameters for functions, note that for some functions (see functions list), parameter#5 (N_x3_x4) is absent. If it is absent, parameters described as #6, #7 stay parameters #5, #6.

Parameters List.

1. **PointerA.** Pointer to start input A macro column.
2. **PointerB.** Pointer to start input B block.
3. **PointerC.** Pointer to start output C macro column.
4. **PointerPB.**
   Pointer for B-prefetch, used when last B block executed. For all internal iterations, function generates prefetch address internally, this pointer used for last iteration only, when function "don't know" address of next B block.
5. **N_x3_x4.**

   Counter of unrolled cycle for calculation length of line, see formulas at functions description.
6. **Ha.**

   Number of lines in the A macro column.
7. **Nb.**
   Number of B blocks per series, for multiply by A macro column.

```
// Matrix multiplication, variant without cycle, for number of atoms not above 3(4)
PROCEDURE <Name>
        ( PointerA  : P_Float64;    // Pointer to matrix A, aligned by 32
          PointerB  : P_Float64;    // Pointer to matrix B, aligned by 32
          PointerC  : P_Float64;    // Pointer to matrix C, aligned by 32
          PointerPB : P_Float64;    // Pointer for B-prefetch
          Ha        : Uint32;       // Lines per A macro column
          Nb        : Uint32);      // B blocks per series
```

<Name> names of functions:

MultiplyMatrixBig_Wb1_1 , MultiplyMatrixBig_Wb1_2 , MultiplyMatrixBig_Wb1_3, MultiplyMatrixBig_Wb1_4.

MultiplyMatrixBig_Wb2_1 , MultiplyMatrixBig_Wb2_2 , MultiplyMatrixBig_Wb2_3, MultiplyMatrixBig_Wb2_4.

MultiplyMatrixBig_Wb3_1 , MultiplyMatrixBig_Wb3_2 , MultiplyMatrixBig_Wb3_3

Decode names of functions. *MultiplyMatrixBig_WbX_Y.*

X = numbers of lines per B matrix block.

Y = length of line in atoms.

```
// Matrix multiplication, variant with cycle, for numbers of atoms above 3(4)
PROCEDURE <Name>
    ( PointerA  : P_Float64;   // Pointer to matrix A, aligned by 32
      PointerB  : P_Float64;   // Pointer to matrix B, aligned by 32
      PointerC  : P_Float64;   // Pointer to matrix C, aligned by 32
      PointerPB : P_Float64;   // Pointer for B-prefetch
      N_x3_x4   : Uint32;      // Counter for line length definition
      Ha        : Uint32;      // Lines per A macro column
      Nb        : Uint32);     // B blocks per series
```

<Name> names of functions:

MultiplyMatrixBig_Wb1_4N1 , MultiplyMatrixBig_Wb1_4N2 , MultiplyMatrixBig_Wb1_4N3, MultiplyMatrixBig_Wb1_4N4.

MultiplyMatrixBig_Wb2_4N1 , MultiplyMatrixBig_Wb2_4N2 , MultiplyMatrixBig_Wb2_4N3, MultiplyMatrixBig_Wb2_4N4.

MultiplyMatrixBig_Wb3_3N1 , MultiplyMatrixBig_Wb3_3N2 , MultiplyMatrixBig_Wb3_3N3

Decode names of functions. *MultiplyMatrixBig_WbX_YNZ.*

X = number of lines per B matrix block.

Y = length of one unrolled cycle iteration in atoms.

Z = addend for calculation length of line.

Length of line calculated by formula:

Hb = N_x3_X4 * Y + Z

In this formula N_x3_x4 - input parameter of function, Y,Z - numbers from name of function.

For example, for setup B-blocks size is 3 lines when line length is 48 atoms, required set input parameter N_x3_x4 = 15 and call function *MultiplyMatrixBig_Wb3_3N3*.

Hb = 15*3+3 = 45+3 = 48.

Functions group, used Base Cycle Version 1 (named - horizontal add method) built as system of nested cycles.

1) <u>*Basic Cycle*</u>. Multiply one A line by one B line. This operation generates C atom. C atom stored as addition to memory (not write memory only), because single pass generates one addend only. This method requires series of "horizontal additions" with AVX registers before store result. Basic cycle contain 3 parts: *prefix*, *cycle* and *tail*. Number of executed atoms equal number of cycle iterations +1, because prefix and tail calculates one atom. Cycle unrolled by 3 or 4 depends on function. Optimization for given line length execution by the unrolled cycle

provided by set of similar function variants selected by high level program by speed maximizing criteria. For detail description of Base Cycle see document *Basic_Cycle_2x4_1-1b.doc*.

2) Cycle for multiplication series of B lines by one A line *(L1 cycle)*, this cycle is external cycle for the basic cycle (item 1). For each next series of basic cycle iterations, changed address of B line. Address of A line is constant, because this cycle works with one A line. Address of result matrix C increments by one atom after each series of basic cycle iterations.

3) Cycle for series of A lines *(L2 cycle)*. Before start of this iterations series restore pointer for B, for repeat work with B block. Modify pointer for A macro column for select next line of A macro column.

4) Cycle for series of B blocks. Each iteration of this cycle works with next B block. Pointer A is const and restored before each iteration for repeat operation with A macro column.

For minimizing delays (latency) when access to matrices data, read-ahead technology used. Data loaded to cache memory by special instructions (Prefetch Hints), this instructions is available in the SSE/AVX instruction set extensions. Main rule for prefetch frequency select is: we must load next block at time of execution current block. This depends from matrix parameters ratio: Hb (line length), Wb (number of lines per B block), Ha (number of lines per A macro column). High level program must consider this fact when select sizes of lines and blocks. Existence of different functions variants for different Wb values is due to this fact.

Prefetch address generation logic assume special cases when next address cannot be generated as current address + step. This includes operation repeats for same block and last operation with current blocks series. First case handled internally, prefetch address returned to start address of block. Second case means use external pointer, passed to function by high level program as function input parameter.

For Intel Sandy Bridge Architecture, one prefetch operation loads 64 bytes to cache memory. Because hardware prefetch works parallel with software prefetch, in some cases prefetch can load 128 bytes or larger.

Frequency of next line prefetches from A macro column selected by criteria: next A line must be loaded during current A line executed. This time means time of B block lines series execution. This means:

1 prefetch for each atom if Wb=1
1 prefetch for each second atom if Wb=2
1 prefetch for each third atom if Wb=3

Frequency of B blocks prefetches is constant for all functions - one prefetch for series of basic cycle iterations. At time for current B block executed (time for A macro column executed) next B block must be loaded. This means number of A macro column lines (Ha) must be equal to number of atoms per line (Hb). Note about automatically dependency from Wb parameter, because number of prefetch operations grow proportionally Wb grow.

Prefetch for output matrix C not used, it disabled by benchmarks results for Basic Cycle with latent output write. See details at document *Basic_Cycle_2x4_1-1b.doc* for details about basic cycle.

3. Group of functions for matrix multiplication by Basic Cycle Version 2 (for small matrices, broadcast schematics).

- Main Calculation Procedure (MCP) for operation «Matrix Multiplication» → multiplication one A floor by sequence of blocks one vertical B section. (illustrated in pictures Mutr_Mult_V2_algorithm.GIF).
- All pointers is 64 bit, aligned by 32 bytes (5 low bits of pointers must be zero).
- Length of line (Wb) selected by select one function from group of similar functions, counter of unrolled cycle iterations and addend. See formula in the functions descriptions.
- Number of lines per B block (Hb) selected by select one of function from group of similar functions, it not pass as explicit parameter. See decode for functions names.
- All counters is 32 bit integer.
- A-atom contains 2 quads of numbers (64 bytes).
- B-atom contains 4 quads of numbers (128 bytes).
- Store to output matrix C executed by addition results to memory (not write data only). This means matrix C must be cleared before operation, if addition to exist data is not required.
- When prepare input parameters for functions, note about for some functions (see functions list), parameter#5 (N_x3_x4) absent. If it absent, parameters described as #6, #7 stay parameters #5, #6.

List of parameters.

1. **PointerA.** Pointer to start input A macro column.
2. **PointerB.** Pointer to start input B block.
3. **PointerC.** Pointer to start output C block.
4. **PointerPB.**
   This pointer used for B prefetch when last B block executed. For all internal iterations, function generates prefetch address internally, this pointer used for last iteration only, when function "don't know" address of next B block.
5. **N_x3_x4.**

   Counter for unrolled cycle iterations, used for calculation length of line, see formula in the functions description.
6. **Ha.**

   Number of lines per A macro column.

7. **Nb.**
   Number of B blocks in the series, note this series multiply by A macro column.

```
// Matrix multiplication, variant without cycle, for number of atoms not above 3(4)
PROCEDURE <Name>
        ( PointerA   : P_Float64;   // Pointer to matrix A, aligned by 32
          PointerB   : P_Float64;   // Pointer to matrix B, aligned by 32
          PointerC   : P_Float64;   // Pointer to matrix C, aligned by 32
          PointerPB  : P_Float64;  // Pointer for B-prefetch
          Ha         : Uint32;        // Lines per C blocks
          Nb         : Uint32);       // B blocks per series
```

<Name> names of functions:
MultiplyMatrixSmall_Hb1_1 , MultiplyMatrixSmall_Hb1_2 , MultiplyMatrixSmall_Hb1_3, MultiplyMatrixSmall_Hb1_4.
MultiplyMatrixSmall_Hb2_1 , MultiplyMatrixSmall_Hb2_2 , MultiplyMatrixSmall_Hb2_3, MultiplyMatrixSmall_Hb2_4.
MultiplyMatrixSmall_Hb3_1 , MultiplyMatrixSmall_Hb3_2 , MultiplyMatrixSmall_Hb3_3

Decode names of functions. *MultiplyMatrixSmall_HbX_Y.*

X = number of lines per B block.

Y = line length in atoms.

```
// Matrix multiplication, variant with cycle, for number of atoms above 3(4)
PROCEDURE <Name>
        ( PointerA   : P_Float64;   // Pointer to matrix A, aligned by 32
          PointerB   : P_Float64;   // Pointer to matrix B, aligned by 32
          PointerC   : P_Float64;   // Pointer to matrix C, aligned by 32
          PointerPB  : P_Float64;  // Pointer for B prefetch
          N_x3_x4    : Uint32;      // Counter for atoms in line
          Ha         : Uint32;        // Lines per C block
          Nb         : Uint32);       // B blocks per series
```

<Name> names of functions:
MultiplyMatrixSmall_Hb1_4N1 , MultiplyMatrixSmall_Hb1_4N2 , MultiplyMatrixSmall_Hb1_4N3, MultiplyMatrixSmall_Hb1_4N4.
MultiplyMatrixSmall_Hb2_4N1 , MultiplyMatrixSmall_Hb2_4N2 , MultiplyMatrixSmall_Hb2_4N3, MultiplyMatrixSmall_Hb2_4N4.
MultiplyMatrixSmall_Hb3_3N1 , MultiplyMatrixSmall_Hb3_3N2 , MultiplyMatrixSmall_Hb3_3N3

Decode names of functions. *MultiplyMatrixSmall_HbX_YNZ.*

X = number of lines per B block.

Y = number of one unrolled iteration in atoms.

Z = addend for calculation line length.

Line length can be calculated by formula:

Wb = N_x3_X4 * Y + Z

In this formula N_x3_x4 - input parameter of function, Y,Z - numbers from function name.

For example, for setup B block size 1 line when line length is 17 atoms, required set input parameter N_x3_x4 = 4 and call function *MultiplyMatrixSmall_Hb1_4N1*.

Wb = 4*4+1 = 16+1 = 17.

Functions group, used Base Cycle Version 2 (named broadcast method) built as system of nested cycles.

1) *Basic Cycle*. Multiply one A atom (this atom under BroadCast to 8 AVX-registers) by B line. This operation generates C line. C line stored to memory as addition to memory (not write only), because each pass generates one addend only. Basic cycle contain *prefix*, *cycle* and *tail*. Number of executed atoms equal number of cycle iterations +1, because prefix and tail calculates one atom. Cycle unrolled by 3 or by 4 depends from function. Speed optimization must be provided in the high level program by selection one of function in the group of similar functions. See document *Basic_Cycle_2x4_2_LAST.doc* for detail description.

2) Cycle for series of B lines *(L1 cycle)*, this cycle is external cycle for the basic cycle (item 1). For each next series of basic cycle iterations, loads new values of A-broadcast and go to next B line. Address of C line is restored, because basic cycle executed for same C line.

3) Cycle for series of C lines *(L2 cycle)*. At start this series of iterations, restore B pointer for repeat B block execution and modify C pointer, means go to next line. Addressing of A matrix and broadcast loads advance to next data.

4) Cycle for series of B blocks. Each iteration of this cycle works with next B block. C pointer restored before each iteration for repeat work with current C block. Addressing of A matrix and broadcast loads advance to next data. This cycle works with one full horizontal floor of A matrix one full vertical section of B matrix. This procedure generates one C block and can be executed as one function call if all blocks has same size in the A horizontal floor and all blocks has same size in the B vertical section.

For minimizing delays (latency) when access to matrices data, read-ahead technology used. Data loaded to cache memory by special instructions (Prefetch Hints), this instructions is available in the SSE/AVX instruction set extensions. Main rule for prefetch frequency select is: we must load next block at time of execution current block. This depends from matrix parameters ratio: Wb (length of line), Hb (number of lines in the B block), Ha (number of lines in the C block). High level program must consider this fact when select sizes of lines and blocks. Existence of different functions variants for different Wb values is due to this fact.

Prefetch address generation logic assume special cases when next address cannot be generated as current address + step. This includes operation repeats for same block and last operation with current blocks series. First case handled internally, prefetch address returned to start address of block. Second case means use external pointer, passed to function by high level program as function input parameter.

For Intel Sandy Bridge Architecture, one prefetch operation loads 64 bytes to cache memory. Because hardware prefetch works parallel with software prefetch, in some cases prefetch can load 128 bytes or larger.

For input matrix A, next atom prefetch executed after load current atom.

Frequency of B block prefetches is constant for all functions - one prefetch for one C line.

This parameter under optimization and can be changed in the next versions of library.

Frequency of next line prefetches from macro column of output matrix C selected by criteria: next C line must be loaded during execution current C line. This means time of execution series of B block lines. This means:

1 prefetch for each atom if Hb=1
1 prefetch for each second atom if Hb=2
1 prefetch for each third atom if Hb=3.

Note.

For algorithm with Basic Cycle V1 (horizontal addition method), line of B block is vertical, this means Hb parameter defines number of basic cycle iterations (line length). Wb defines number of lines (block B height).

For algorithm with Basic Cycle V2 (broadcast method), line of B block is horizontal, this means Wb parameter defines number of basic cycle iterations (line length). Hb defines number of lines (block B height).

4. Pack matrix A macro column (variant for big matrices).

- Pack one macro column of matrix A (illustration picture at file *Repack_A.GIF*).
- A-atom contains 2 quads of numbers from 2 adjacent lines and packed to linear sequence of 8 numbers.
- All pointers is 64 bit, aligned by 32 bytes (5 low bits is zero).
- All counters is 32 bit unsigned integers.

List of parameters.

1. **PointerA**. Pointer to A macro column.
2. **PointerBuffer**. Pointer to transit buffer.
3. **Xa**.
   Size of matrix A one line, bytes. This value adds to pointer for addressing

next line. It must be aligned by 32, alignment required same as for pointers.

4. **Hb**. Number of atoms in the line.
5. **Ha**. Number of lines in the A macro column.
6. **FlagTransp.**
   Transpose flag for matrix A. Zero value (FALSE) means A-matrix a is not transposed, non-zero value (TRUE) means A-matrix is transposed.

```
void ( __stdcall *PackA )
( double* PointerA,  double* PointerBuffer,
long uint Xa, long uint Hb, long uint Ha, long uint FlagTransp );
```

```
// Pack A matrix macro column
PROCEDURE PackA
   ( PointerA       : P_Float64;    // Pointer to A macro column, align 32
     PointerBuffer : P_Float64;     // Pointer to buffer, align 32
     Xa            :  Uint32;       // Matrix A line length, bytes
     Hb            :  Uint32;       // Number of atoms per line
     Ha            :  Uint32;       // Number of lines
     FlagTransp     : Uint32 );      // Transpose flag
```

5. Pack matrix B block (variant for big matrices).

- Pack one block of matrix B (illustration picture at file *Repack_B.GIF*).
- B-atom contains 4 quads of numbers from 4 adjacent columns and packed to linear sequence of 16 numbers.
- All pointers is 64 bit, aligned by 32 bytes (5 low bits is zero).
- All counters is 32 bit unsigned integers.

List of parameters.

1. **PointerB**. Pointer to source B block.

2. **PointerBuffer**. Pointer to transit buffer.

3. **Xb**.
   Size of matrix B one line. Function uses this value for vertical movements in the matrix. For shift by N strings down, value Xb*N must be added to pointer. Because this value added to pointer, it must be aligned by 32, alignment required same as for pointers, pointer must be aligned after addition also.

4. **Hb**. Number of atoms per line.

5. **Wb**. Number of lines per B block.

```
void ( __stdcall *PackB )
( double* PointerB,  double* PointerBuffer,
long uint Xb, long uint Hb, long uint Wb );
```

```
// Pack matrix B block
PROCEDURE PackB
   ( PointerB        : P_Float64;     // Pointer to block B, aligned by 32
     PointerBuffer   : P_Float64;     // Pointer to buffer, aligned by 32
     Xb              :  Uint32;       // Length of B line, bytes
     Hb              :  Uint32;       // Number of atoms per line
     Wb              :  Uint32 );     // Number of lines
```

6. Unpack matrix A macro column (variant for big matrices)

- Restore one macro column of matrix A.
- A-atom contains 2 quads of numbers from 2 adjacent lines.
- All pointers is 64 bit, aligned by 32 bytes (5 low bits must be zero).
- All counters is unsigned integers.

Function provides restore one given macro column of matrix A, used for restore original matrix state after pack. This operation is reverse to PackA.

List of parameters.

1. **PointerA**. Pointer to A macro column, previously packed by function PackA.

2. **PointerBuffer**. Pointer to transit buffer, used for restore macro columns.

3. **Xa**.

   Size of matrix A line in bytes. This value added to pointer for go to next line and must be aligned by 32 same as pointers, because pointer must be aligned after addition also.

4. **Hb**. Number of atoms per line.

5. **Ha**. Number of lines per one macro column.

*void (__stdcall *UnPackA)*
(double PointerA, double* PointerBuffer,*
long uint Xa, long uint Hb, long uint Ha, ~~long int FlagTransp~~);

```
// Restore matrix A macro column
PROCEDURE UnPackA
   ( PointerA        : P_Float64;    // Pointer to A macro column, aligned by 32
     PointerBuffer : P_Float64;      // Pointer to buffer, aligned by 32
     Xa               :  Uint32;        // Matrix A line length, bytes
     Hb               :  Uint32;        // Atoms per line
     Ha               :  Uint32 );      // Number of lines
```

7. Unpack matrix B block (variant for big matrices).

- Restore one block of matrix B.
  B-atom contains 4 quads of numbers from 4 adjacent columns.
- All pointers is 64 bit, aligned by 32 bytes (5 low bits must be zero).
- All counters is unsigned integers.

Function provides restore one given block of matrix B, used for restore original matrix state after pack. This operation is reverse to *PackB*.

List of parameters.

1. PointerB. Pointer to B block, previously packed by function PackB.

2. PointerBuffer. Pointer to transit buffer, used for restore blocks.

3. Xb.
Size of matrix B line in bytes. This value added to pointer for go to the next line and must be aligned by 32 same as pointers, because pointer must be aligned after addition also.

4. Hb. Number of atoms per line.

5. Wb. Number of lines per B block.

```
void ( __stdcall *UnPackB )
( double* PointerB,  double* PointerBuffer,
long uint Xb, long uint Hb, long uint Wb );


// Restore matrix B block
PROCEDURE UnPackB
   ( PointerB         : P_Float64;     // Pointer to block B, aligned by 32
     PointerBuffer    : P_Float64;     // Pointer to buffer, aligned by 32
     Xb               :  Uint32;       // Matrix B line length, bytes
     Hb               :  Uint32;       // Number of atoms per line
     Wb               :  Uint32 );     // Number of lines
```

8. Unpack matrix C macro column (variant for big matrices).

For this operation used function *UnPackA*, described above. Function called with appropriate macro columns parameters. (Illustration in the file *Repack_C.GIF*). By this reason, function *UnPackC* absent in the library.

1. Floor copied to the buffer. C_PackedMcrC^ located in the buffer with linear access to elements.
2. Atom C size is 2x4: "atom height A" x "atom width B".
   With used sizes of atoms A-atom (2x4) and B-atom (4x4) size of C-atom is same as size of A-atom. With used algorithm of calculations, C macro column generated with same structure as A macro column.
3. Example, for hypothetical variant with other sizes: size of A-atom (4x4) and B-atom (4x2), structure of matrix C is different and required separate function *UnPackC*.

9. Pack matrix A block (variant for small matrices).

- Function packs one A-block. High level program must pack series of A blocks by call this function for each block. Block packed to the transit buffer. (Illustration at files *MxM_BroadCast_Repack.DOC*, *Mutr_Mult_V2_algorithm.GIF*, *Repack_2_A.GIF*).
- All pointers is 64 bit, aligned by 32 bytes (5 low bits is zero).
- All counters is 32 bit unsigned.
- A-atom contains 2 quads of numbers (64 bytes).

List of Parameters.

1. **PointerA.** Pointer to start input A block.
2. **PointerBuffer.** Pointer to transit buffer.
3. **Xa.**
   Matrix A line size in bytes. This value added to pointer for go to the next line. This value must be aligned by 32, same as pointers, because pointer must be aligned by 32 after addition also.
4. **Hb**. Number of atoms in the line.
5. **Ha.** Number of lines per A block.
6. **FlagTransp.**
   Matrix A transpose flag. Zero value (FALSE) means input matrix A is not transposed. Non-zero value (TRUE) means input matrix A is transposed.

```
// Matrix A repack
PROCEDURE PackASmall
        ( PointerA       : P_Float64;    // Pointer to matrix A, aligned by 32
          PointerBuffer  : P_Float64;    // Pointer to buffer, aligned by 32
          Xa             : Uint32;       // Bytes per one full line of matrix A
          Hb             : Uint32;       // Atoms per line
          Ha             : Uint32;       // Lines per A block
          FlagTransp     : Uint32);      // Matrix A transpose flag
```

10. Pack matrix B block (variant for small matrices).

- Function packs one B-blocks. High level program must pack series of B blocks by call this function for each block. Block packed to the transit buffer. (Illustration at files *MxM_BroadCast_Repack.DOC*, *Mutr_Mult_V2_algorithm.GIF*, *Repack_2_B.GIF*).
- All pointers is 64 bit, aligned by 32 bytes (5 low bits is zero).
- All counters is 32 bit unsigned.
- B-atom contains 4 quads of numbers (128 bytes).

List of Parameters.

1. **PointerB.** Pointer to start input B block.
2. **PointerBuffer.** Pointer to transit buffer.
3. **Xb.**
   Matrix B line size in bytes. This value added to pointer for go to the next line. This value must be aligned by 32, same as pointers, because pointer must be aligned by 32 after addition also.
4. **Wb.** Number of atoms per line.
5. **Hb.** Number of lines per B block.

```
// Matrix B repack
PROCEDURE PackBSmall
        ( PointerB        : P_Float64;     // Pointer to matrix B, aligned by 32
          PointerBuffer   : P_Float64;     // Pointer to buffer, aligned by 32
          Xb              : Uint32;        // Bytes per one full line of matrix B
          Wb              : Uint32;        // Atoms per line
          Hb              : Uint32);       // Lines per B block
```

11. Unpack matrix A block (variant for small matrices).

- Function unpacks one A-block. High level program must unpack series of A blocks by call this function for each block. Block unpacked to the transit buffer. (Illustration at files *MxM_BroadCast_Repack.DOC*, *Mutr_Mult_V2_algorithm.GIF*, this operation is reverse to operation show at file *Repack_2_A.GIF*).
- All pointers is 64 bit, aligned by 32 bytes (5 low bits must be zero).
- All counters is 32 bit unsigned.
- A-atom contains 2 quads of numbers (64 bytes).

List of parameters.

1. **PointerA.** Pointer to start input A block.
2. **PointerBuffer.** Pointer to transit buffer.
3. **Xa.**
   Matrix A line size in bytes. This value added to pointer for go to the next

line. This value must be aligned by 32, same as pointers, because pointer must be aligned by 32 after addition also.

4. **Hb.** Number of atoms per line.
5. **Ha.** Number of lines per A block.

```
// Matrix A unpack
PROCEDURE UnPackASmall
        ( PointerA        : P_Float64;     // Pointer to matrix A, aligned by 32
          PointerBuffer   : P_Float64;     // Pointer to buffer, aligned by 32
          Xa              : Uint32;        // Bytes per one full line of matrix A
          Hb              : Uint32;        // Atoms per line
          Ha              : Uint32);       // Lines per A block
```

12. Unpack matrix B block (variant for small matrices).

- Function unpacks one B-block. High level program must unpack series of B blocks by call this function for each block. Block unpacked to the transit buffer. (Illustration at files *MxM_BroadCast_Repack.DOC*, *Mutr_Mult_V2_algorithm.GIF*, this operation is reverse to operation show at file *Repack_2_B.GIF*).
- All pointers is 64 bit, aligned by 32 bytes (5 low bits must be zero).
- All counters is 32 bit unsigned.
- B-atom contains 4 quads of numbers (128 bytes).

List of parameters.

1. **PointerB.** Pointer to start input B block.
2. **PointerBuffer.** Pointer to transit buffer.
3. **Xb.**
   Matrix B line size in bytes. This value added to pointer for go to the next line. This value must be aligned by 32, same as pointers, because pointer must be aligned by 32 after addition also.
4. **Wb.** Number of atoms per line.
5. **Hb.** Number of lines per B block.

```
// Matrix B unpack
PROCEDURE UnPackBSmall
        ( PointerB        : P_Float64;     // Pointer to matrix B, aligned by 32
          PointerBuffer   : P_Float64;     // Pointer to buffer, aligned by 32
          Xb              : Uint32;        // Bytes per one full line of matrix A
          Wb              : Uint32;        // Atoms per line
          Hb              : Uint32);       // Lines per B block
```

13. Unpack output matrix C (variant for small matrices).

Because output matrix C restored (unpacked) «by floors», algorithm of this function is equivalent to unpack input matrix A, with different length of line, controlled by input parameter of function. This means, already existed function UnPackASmall can be used for unpack matrix C, no new function required for this operation. Illustrations in the documents *MxM_BroadCast_Repack.DOC*, *Mutr_Mult_V2_algorithm.GIF*, *Repack_2_A.GIF*.

Appendix 1. Compatibility notes.

This library requires processor with support AVX instruction set. Additionally, for save and restore 256-bit context of processor, required support AVX technology by operating system.

Before use functions of this library, high level program must check AVX support. Additional library *AVX_OS.DLL* contain functions, required for this verifications. For select optimal sizes of data blocks, high level program can use cache memory parameters, returned by functions of library *AVX_OS.DLL*.

Remember about virtualization technologies. Virtual machine can reduce processor functionality, some instructions set, supported by physical CPU, can be disabled in the virtual CPU.

Reverse situation also possible. Virtual CPU can emulate functionality, not supported by physical CPU. But in this situation performance speed is too low.

Appendix 2. Interface notes.

For calculation and pack-unpack functions (but not for system information functions in other libraries) all pointers must be aligned by vector register length if other not specified. For lengths of data blocks, use units equal numbers per vector register. For example, 256-bit AVX technology means alignment factor equal 32 bytes, numbers per register is 4 numbers (double precision numbers, 64 bit).

Library interface compatible with *Microsoft x64 calling convention.*

Integer 64-bit parameters passed as shown:

1 RCX
2 RDX
3 R8
4 R9

5 stack [RSP+40]
6 stack [RSP+48]
7 stack [RSP+56]
...continued in the stack frame

Floating point numbers passed as shown (one number in the low bits of each register):

1 XMM0
2 XMM1
3 XMM2
4 XMM3
5 stack [RSP+40]
6 stack [RSP+48]
7 stack [RSP+56]
... continued in the stack frame

For processor registers used next rule: If, for example, first parameter is integer and second - floating point, that means first parameter passed in RCX, second in XMM1. XMM0 skipped. If, for example first parameter is floating point, second - integer, first parameter passed in XMM0, second in RDX. RCX skipped.

For parameters (1-4) stack space already reserved and must allocated by caller (this space named parameters shadow), however parameters (1-4) passed in the registers, not in the stack.

Parameters addresses in the stack [RSP+40], [RSP+48], [RSP+56]... calculated for RSP value, already modified by function subroutine call. Save 64-bit instruction pointer *RIP* in the stack cause stack pointer *RSP* decreased by 8.

Status code or *return code* (if used) returned in the RAX register.

Four (or one by classic convention) floating point parameters returned in the registers XMM0-XMM3 (one number in the low bits of each register).

Function can destroy this volatile registers:
RAX, RCX, RDX, R8-R11, XMM0-XMM5.

Function can't destroy (must save and restore if use) this non-volatile registers:
RBX, RBP, RDI, RSI, R12-R15, XMM6-XMM15

High 128-bit halves of all 256-bit registers YMM0-YMM15 can be destroyed by function. For performance optimization, recommended clear this fields by *VZEROUPPER* instruction to minimize penalty when switch between AVX and SSE code.

Stack pointer value (RSP) must be aligned by 16 bytes before call function subroutine.