Disclaimer and Legal Information

INFORMATION IN THIS DOCUMENT IS PROVIDED IN CONNECTION WITH INTEL PRODUCTS. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. EXCEPT AS PROVIDED IN INTEL'S TERMS AND CONDITIONS OF SALE FOR SUCH PRODUCTS, INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO SALE AND/OR USE OF INTEL PRODUCTS INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

A "Mission Critical Application" is any application in which failure of the Intel Product could result, directly or indirectly, in personal injury or death. SHOULD YOU PURCHASE OR USE INTEL'S PRODUCTS FOR ANY SUCH MISSION CRITICAL APPLICATION, YOU SHALL INDEMNIFY AND HOLD INTEL AND ITS SUBSIDIARIES, SUBCONTRACTORS AND AFFILIATES, AND THE DIRECTORS, OFFICERS, AND EMPLOYEES OF EACH, HARMLESS AGAINST ALL CLAIMS COSTS, DAMAGES, AND EXPENSES AND REASONABLE ATTORNEYS' FEES ARISING OUT OF, DIRECTLY OR INDIRECTLY, ANY CLAIM OF PRODUCT LIABILITY, PERSONAL INJURY, OR DEATH ARISING IN ANY WAY OUT OF SUCH MISSION CRITICAL APPLICATION, WHETHER OR NOT INTEL OR ITS SUBCONTRACTOR WAS NEGLIGENT IN THE DESIGN, MANUFACTURE, OR WARNING OF THE INTEL PRODUCT OR ANY OF ITS PARTS.

Intel may make changes to specifications and product descriptions at any time, without notice. Designers must not rely on the absence or characteristics of any features or instructions marked "reserved" or "undefined". Intel reserves these for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from future changes to them. The information here is subject to change without notice. Do not finalize a design with this information.

The products described in this document may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request.

Contact your local Intel sales office or your distributor to obtain the latest specifications and before placing your product order.

Copies of documents which have an order number and are referenced in this document, or other Intel literature, may be obtained by calling 1-800-548-4725, or go to:

Intel processor numbers are not a measure of performance. Processor numbers differentiate features within each processor family, not across different processor families. Go to:

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products.

Intel, Intel logo, Intel Core, VTune, Xeon are trademarks of Intel Corporation in the U.S. and other countries.

* Other names and brands may be claimed as the property of others.

OpenCL and the OpenCL logo are trademarks of Apple Inc. used by permission from Khronos.

Microsoft product screen shot(s) reprinted with permission from Microsoft Corporation.

Copyright © 2013 Intel Corporation. All rights reserved.
Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804
## Contents

About General Matrix Multiply sample ................................................................. 5  
Path ......................................................................................................................... 5  
Algorithm ................................................................................................................ 5  
OpenCL* Implementation ....................................................................................... 6  
Understanding the OpenCL Performance Characteristics ..................................... 10  
  Benefits of Compiler Implicit Vectorization ......................................................... 10  
  Work-group Size Considerations ......................................................................... 10  
Project Structure ................................................................................................... 10  
APIs Used ................................................................................................................ 11  
Reference (Native) Implementation ...................................................................... 11  
Controlling the Sample .......................................................................................... 12
**About General Matrix Multiply sample**

General Matrix Multiply (GEMM) sample demonstrates how to efficiently utilize an OpenCL* device to perform general matrix multiply operation on two dense square matrices. The primary target devices that are suitable for this sample are the devices with cache memory: Intel® Xeon Phi™ and Intel® Architecture CPU OpenCL devices. This implementation optimizes trivial matrix multiplication nested loop to utilize the memory cache more efficiently by introducing a well-known practice as tiling (or blocking), where matrices are divided into blocks and the blocks are multiplied separately to maintain better data locality.

**Path**

<table>
<thead>
<tr>
<th>Location</th>
<th>Executable</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;INSTALL_DIR&gt;/GEMM</td>
<td>gemm</td>
</tr>
</tbody>
</table>

**Algorithm**

General Matrix Multiply is a subroutine that performs matrix multiplication:

\[
C := \alpha A \ast B + \beta C,
\]

where \(A, B\) and \(C\) are dense matrices and \(\alpha\) and \(\beta\) are floating point scalar coefficients. The sample supports single-precision and double-precision data types for matrix elements (as well as \(\alpha\) and \(\beta\) constants).

Matrix \(A\) and matrix \(B\) may come in the transposed or regular layouts. From the implementation point of view, the transposition of a matrix can be interpreted just as switch between the storage methods:

- row-major
- column-major.

This sample supports two modes:

1. Normal-normal (NN), where \(A, B\) and \(C\) matrices are stored in the column-major order.
2. Normal-transposed (NT), where \(A\) and \(C\) matrices are stored in column-major order, and the \(B\) matrix is stored in the row-major order.

This matrix multiplication appears as the following pseudo-code (the NN variant for square matrices of a given size):

```plaintext
for i from 0 to size-1
    for j from 0 to size-1
        c = 0
        for k from 0 to size-1
            c = c + A(k, i)*B(j, k)
        end for
        C(j, i) = alpha*c + beta*C(j, i)
    end for
```

```
Now consider the tiled (or blocked) matrix multiplication as an optimization technique that improves data reuse on architecture where at least two memory hierarchies exist:

- Slow and big main memory
- Fast but small memory.

For the Intel CPUs and the Intel Xeon Phi coprocessor devices the fast memory is the regular cache memory (data L1/L2), or regular CPU registers. To utilize it you need to maximize data reuse, so instead of producing one resulting element of matrix C in the inner loop above (loop by the k index), now calculate a block. For example:

```plaintext
for i from 0 to NUM_OF_TILES_M-1
  for j from 0 to NUM_OF_TILES_N-1
    C_BLOCK = ZERO_MATRIX(TILE_SIZE_M, TILE_SIZE_N)
    for k from 0 to size-1
      for ib = from 0 to TILE_SIZE_M-1
        for jb = from 0 to TILE_SIZE_N-1
          C_BLOCK(jb, ib) = C_BLOCK(ib, jb) + A(k, i*TILE_SIZE_M + ib)*B(j*TILE_SIZE_N + jb, k)
        end for
      end for
    end for
  end for
end for

for ib = from 0 to TILE_SIZE_M-1
  for jb = from 0 to TILE_SIZE_N-1
    C(j*TILE_SIZE_M + jb, i*TILE_SIZE_N + ib) = C_BLOCK(jb, ib)
  end for
end for
```

**OpenCL* Implementation**

General Matrix Multiplication sample implements the “blocked” variant of matrix multiplication. The `gemm.cl` file consists of the following kernels:

- `gemm_nn`
- `gemm_nt`

The difference is in the format of matrix B as explained in the “Algorithm” section. Kernels are executed on a two-dimensional iteration space (NDRange). The global size for the dimensions is:

0. `size/TILE_SIZE_M` for the zero dimension of NDRange (`get_global_size(0)`)  
1. `size/TILE_SIZE_N` for the first dimension of NDRange (`get_global_size(1)`)
where ‘size’ is the matrix size.

**NOTE:** All matrices are square and equally sized, so there is only one size parameter for all matrices.

Hence \( \text{TILE_SIZE}_M \times \text{TILE_SIZE}_N \) is the number of elements of matrix \( C \), calculated by one work-item in NDRange. This also defines one tile of matrix \( C \), which is calculated from one tile of matrix \( A \) and one tile of matrix \( B \).

To utilize the automatic vectorizer in Intel OpenCL* implementation efficiently and avoid gathers, make all adjacent work-items in dimension 0 read the sequential memory addresses of elements in matrices \( A \) and \( B \), which leads to the implementation, where each work-item processes a tile of the resulting matrix \( C \) in a stridden way. In fact, you should follow this rule in dimension 0 only, but for simplicity and symmetry you can use this rule for both 0 and 1 dimensions.

The following picture illustrates matrix partitioning implemented in the OpenCL NN flavor kernel with \( \text{TILE_SIZE}_M = \text{TILE_SIZE}_N = 2 \) and \( \text{TILE_GROUP}_M = \text{TILE_GROUP}_N = 4 \), where \( \text{TILE_GROUP}_M \) and \( \text{TILE_GROUP}_N \) are work-group size for dimension 0 and 1 correspondingly. In this example work-group sizes are too small to be efficient and used for illustration purposes only. For more information, see the "Work-group Size Considerations" section.
Items, read during one iteration of internal loop of a work-item (along the 0 dimension)

Matrix A

Items, read during one iteration of internal loop for a work-group (along the 0 dimension)

Matrix B

Items, read during one iteration of internal loop of a work-item (along the 1st dimension)

Matrix C

Items, read by the work-group (along the 0 dimension)

Items, read during one iteration of internal loop for a work-group (along the 1st dimension)

Items, read by the work-group (along the 1st dimension)

Memory addresses direction (all matrices in column major order)
An “internal loop” is one of the loops by ib or jb (depending on dimension considered: 0 or 1 correspondingly) in the pseudo-code of the tiled matrix multiplication. Each work-item in this example processes one stridden 2x2 tile reading and writing with the following matrix elements:

A difference between the gemm_nn and the gemm_nt variants of kernels exists. Gemm_nn variant has additional tiling parameter: along the k direction (loop over the k variable in the reference code in the previous section; dot product direction). The matrix B storage has different layout in comparison to the gemm_nt variant and it turns to be useful in performance terms to use blocking along this dimension also. In the source code of the sample this dimension is called the k dimension, and tile size is TILE_SIZE_K.
Understanding the OpenCL Performance

Characteristics

Benefits of Compiler Implicit Vectorization

Selecting proper values for work-group sizes you enable the Intel OpenCL compiler to auto-vectorize the kernel code, which gains performance in comparison to unvectorized version especially on the Intel Xeon Phi coprocessor device, where wide SIMD is used (16 work-items for FP and 8 work-items for DP). So by writing a kernel using scalar data types and proper work-group sizes you still utilize the underlying hardware efficiently.

Work-group Size Considerations

Work-group size in the 0-dimension should be not less than 16 work-items on the Intel Xeon Phi coprocessor device, and 8 work-items on the CPU devices with the Intel® Advanced Vector Extensions (Intel® AVX) support, which enables auto-vectorizer to do its best. It also should be multiple of 16 (or 8 correspondingly) for better performance.

In the same time, work-group size together with the global size determines the number of work-groups running in one clEnqueueNDRange call. Having enough work-groups is a crucial factor for achieving high utilization of the Intel Xeon Phi coprocessor cores. For more details, please refer to Intel SDK for OpenCL Applications - Optimization Guide.

Project Structure

All files, necessary for sample build and execution, reside at the sample directory (GEMM) and in common directory of the root directory to which you extract samples.

GEMM directory contains following files:

- gemm.cpp – GEMM host-side implementation, including the application entry point, validation routine, all OpenCL resources allocation and kernel invocation.
- cmdoptions.hpp, cmdoptions.cpp – sample command-line parameters definition and checking for correct values based on the OpenCL device capabilities.
- gemm.cl – GEMM OpenCL kernels, necessary for correct application run.
• **Makefile** – builds the sample binary.
• **README.TXT** – instruction building and running the sample. Also provides information on understanding the sample output.

## APIs Used

This sample uses the following OpenCL host functions:

- `clGetPlatformIDs`
- `clGetPlatformInfo`
- `clGetDeviceIDs`
- `clGetDeviceInfo`
- `clCreateContext`
- `clCreateCommandQueue`
- `clCreateProgramWithSource`
- `clBuildProgram`
- `clGetProgramBuildInfo`
- `clCreateKernel`
- `clGetKernelWorkGroupInfo`
- `clCreateBuffer`
- `clSetKernelArg`
- `clEnqueueNDRangeKernel`
- `clEnqueueMapBuffer`
- `clEnqueueUnmapMemObject`
- `clFinish`
- `clReleaseMemObject`
- `clReleaseKernel`
- `clReleaseProgram`
- `clReleaseCommandQueue`
- `clReleaseContext`.

## Reference (Native) Implementation

Reference implementation is done in the `checkValidity` routine of the `gemm.cpp` file. This is a single-threaded code, which performs matrix multiplication algorithm in native C++ as described in the “Algorithm” section. Validation is not enabled by default. To enable validation, use the `--validation` command-line switch.
Controlling the Sample

The sample executable is a console application. You can choose platform, device, and other parameters through command line. You can obtain the complete set of parameters by running:

```
gemm -h
```

Under Linux OS it produces the following help text:

Usage:
```
./gemm [OPTIONS]
```

Available options:

- `-h, --help` Show this help text and exit. (Default value: 0)

- `-p, --platform number-or-string`
  Selects the platform, the devices of which are used. (Default value: Intel)

- `-d, --device number-or-string`
  Selects the device on which all stuff is executed. (Default value: 0)

- `-t, --type all | cpu | gpu | acc | default | <OpenCL constant for device type>`
  Selects the device by type on which all stuff is executed. (Default value: all)

- `-s, --size <integer>`
  Size of matrix in elements. (Default value: 3968)

- `-i, --iterations <integer>`
  Number of kernel invocations. For each invocation, performance information will be printed. Zero is allowed: in this case no kernel invocation is performed but all other host stuff is created. (Default value: 10)

- `-a, --arithmetic float | double`
  Type of elements and all calculations. (Default value: float)
--kernel nt | nn
Determines format of matrices involved in multiplication. There are two supported forms: nn and nt; nn is for case when both matrices A and B are in column-major form; nt is for case when A is in column-major form, but B is in row major format (i.e. transposed). Matrices A and C are always in column major format. (Default value: nn)

--validation
Enables validation procedure on host (slow for big matrices). (Default value: 0)

--tile-size-M <integer>
Size of tile for matrix A. (Default value: 1)

--tile-group-M <integer>
Grouping parameter for matrix A. Also defines work group size in 0-dimension. (Default value: 16)

--tile-size-N <integer>
Size of tile for matrix B. (Default value: 128)

--tile-group-N <integer>
Grouping parameter for matrix B. Also defines work group size in 1-dimension. (Default value: 1)

--tile-size-K <integer>
Size of block in dot-product direction (applicable for nn kernel only). (Default value: 8)
References

Intel® SDK for OpenCL® Applications – Optimization Guide