General Matrix Multiply

Sample User's Guide

Intel® SDK for OpenCL Applications - Samples*

Document Number: 329762-004US

Contents

Legal Information

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

About General Matrix Multiply

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.

Algorithm

General Matrix Multiply is a subroutine that performs matrix multiplication:

```
C := alpha*A*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*):

```
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
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:

```
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

            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

      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.

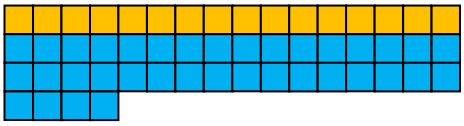
NOTE: All matrices are square and equally sized, so there is only one `size` parameter for all matrices.

Hence `TILE_SIZE_M*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.

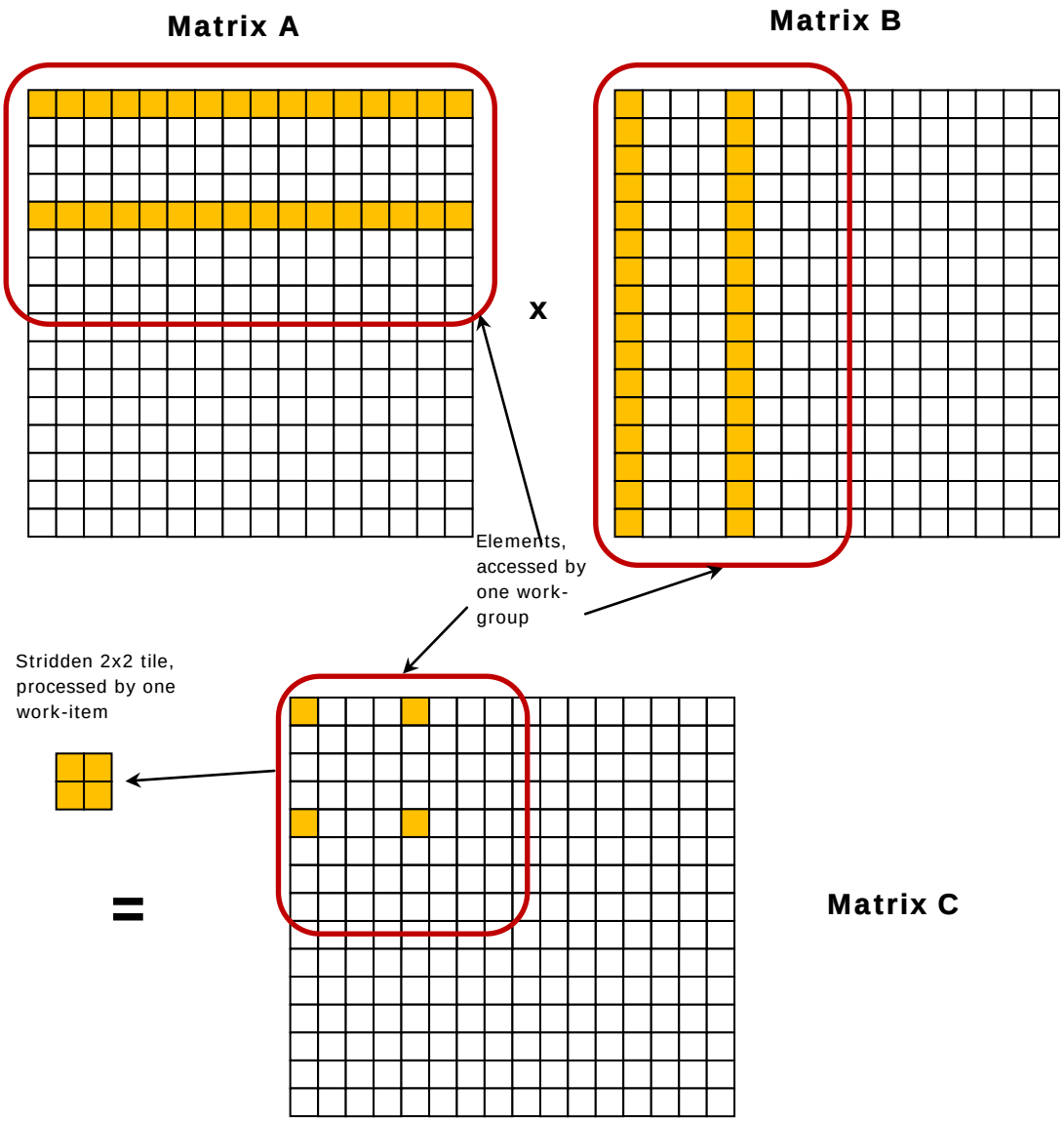
The following picture illustrates matrix partitioning implemented in the OpenCL `NN` flavor kernel with `TILE_SIZE_M = TILE_SIZE_N = 2` and `TILE_GROUP_M = TILE_GROUP_N = 4`, where `TILE_GROUP_M` and `TILE_GROUP_N` are work-group size for dimension 0 and 1 correspondingly. In this example work-group 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)



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

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.

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.

The sample supports the following command-line parameters:

Option	Description
`-h, --help`	Show this help text and exit.
`-p, --platform number-or-string`	Selects the platform, the devices of which are used.
`-t, --type all | cpu | gpu | acc | default | <OpenCL constant for device type>`	Selects the device by type on which the OpenCL kernel is executed.
`-d, --device number-or-string`	Selects the device on which all stuff is executed.
`-s, --size <integer>`	Size of matrix in elements.

`-i, --iterations <integer>`	Number of kernel invocations. For each invoction, performance information will be printed. Zero is allowed: in this case no kernel invocation is performed but all other host stuff is created.
`-a, --arithmetic float | double`	Type of elements and all calculations.
`--kernel nt | nn`	Determines format of matrices involved in multiplication. There are two supported form: 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 (transposed). Matrices A and C are always in column major format.
`--validation`	Enables validation procedure on host (slow for big matrices).
`--tile-size-M <integer>`	Size of tile for matrix A.
`--tile-group-M <integer>`	Grouping parameter for matrix A. Also defines work group size in 0-dimension.
`--tile-size-N <integer>`	Size of tile for matrix B.
`--tile-group-N <integer>`	Grouping parameter for matrix B. Also defines work group size in 1-dimension.
`--tile-size-K <integer>`	Size of block in dot-product direction (applicable for nn kernel only).

Understanding the Sample Output

The following is an example of possible sample output:

```
Platforms (1):
 [0] Intel(R) OpenCL [Selected]
Devices (2):
[0]                 Genuine Intel(R) CPU  @ 2.60GHz [Selected]
[1] Intel(R) Many Integrated Core Acceleration Card
Build program options: "-DT=float -DTILE_SIZE_M=1 -DTILE_GROUP_M=16
-DTILE_SIZE_N=128 -DTILE_GROUP_N=1 -DTILE_SIZE_K=8"
Running gemm_nn kernel with matrix size: 3968x3968
Memory row stride to ensure necessary alignment: 15872 bytes
Size of memory region for one matrix: 62980096 bytes
Using alpha = 0.57599 and beta = 0.872412
Host time: 1.09695 sec.
Host perf: 113.937 GFLOPS
Host time: 1.03923 sec.
Host perf: 120.266 GFLOPS
Host time: 1.09934 sec.
Host perf: 113.69 GFLOPS
Host time: 1.05984 sec.
Host perf: 117.927 GFLOPS
Host time: 1.01676 sec.
Host perf: 122.924 GFLOPS
Host time: 1.04862 sec.
Host perf: 119.189 GFLOPS
```

```
Host time: 0.95813 sec.
Host perf: 130.446 GFLOPS
Host time: 1.02999 sec.
Host perf: 121.345 GFLOPS
Host time: 1.04228 sec.
Host perf: 119.914 GFLOPS
Host time: 1.06426 sec.
Host perf: 117.437 GFLOPS
```

First, the sample outputs all available platforms and picks one of them (look at the line with [Selected]). Then goes the list of devices for the selected platform. The selected device is also marked.

Then follows a "Build program options" section which is exact build options line passed to the `clBuildProgram` OpenCL call.

In the end sample calls the kernel several times and for each iteration it prints two numbers: "Host time" and "Host perf". Host time is time measured on host for complete kernel invocation without any data transfer to/from device. Host perf is the number of GFLOPS calculated based on Host time and the number of floating point operations performed in GEMM kernel (which is easily calculated based on matrix size).

In the case when --validation key is set in the command line, validation procedure prints validation status just after the first kernel iteration. It looks as "PASSED" if validation succeeded and "FAILED" otherwise. Be patient, validation procedure may need a long time to complete even for default arguments.

This is an example of successful execution with validation enabled:

```
Platforms (1):
 [0] Intel(R) OpenCL [Selected]
Devices (2):
[0]                 Genuine Intel(R) CPU  @ 2.60GHz [Selected]
[1] Intel(R) Many Integrated Core Acceleration Card
Build program options: "-DT=float -DTILE_SIZE_M=1 -DTILE_GROUP_M=16
-DTILE_SIZE_N=128 -DTILE_GROUP_N=1 -DTILE_SIZE_K=8"
Running gemm_nn kernel with matrix size: 3968x3968
Memory row stride to ensure necessary alignment: 15872 bytes
Size of memory region for one matrix: 62980096 bytes
Using alpha = 0.57599 and beta = 0.872412
Host time: 1.11617 sec.
Host perf: 111.976 GFLOPS
Validate output... PASSED
Host time: 1.02897 sec.
[redacted]
Host time: 0.936836 sec.
```

References

Intel SDK for OpenCL Applications – Optimization Guide