SGEMM for Intel® Processor Graphics

Contents

Introduction

In this article we are going to demonstrate how to optimize Single precision floating General Matrix Multiply (SGEMM) kernels for the best performance on Intel® Core™ Processors with Intel® Processor Graphics. We implemented our sample using OpenCL and rely heavily on Intel's **cl_intel_subgroups** OpenCL extension. After giving a brief overview of the General Matrix Multiply and **cl_intel_subgroups** extension, we are going to cover our implementation and summarize its performance on 4th and 5th Generation Intel® Core™ Processors with Intel® Processor Graphics.

We want to thank Brijender Bharti, Tom Craver, Ben Ashbaugh, Girish Ravunnikutty, Allen Hux, and Qually Jiang for their help in reviewing this article and the accompanying code.

General Matrix Multiply

From Wikipedia page on Matrix Multiplication: "*In mathematics, matrix multiplication is a binary operation that takes a pair of matrices, and produces another matrix*". Matrix Multiplication is such a common operation with a wide variety of practical applications that it has been implemented in numerous programming languages. Starting from 1979 Basic Linear Algebra Subprograms (BLAS) specification prescribes a common set of routines for performing linear algebra operations, including matrix multiplication. Refer to BLAS Wikipedia page for more details. BLAS functionality has three levels, and here we are going to consider Level 3, which contains matrix-matrix operations of the form:

$$C \leftarrow \alpha AB + \beta C$$

Single float precision General Matrix Multiply (SGEMM) sample we are presenting here shows how to efficiently utilize OpenCL to perform general matrix multiply operation on two dense square matrices. We developed our sample to target Intel® 4th and 5th Generation Processors with Intel® Processor Graphics. Our implementation relies on Intel's **cl_intel_subgroups** extension to OpenCL to optimize matrix multiplication for more efficient data sharing.

cl_intel_subgroups Extension

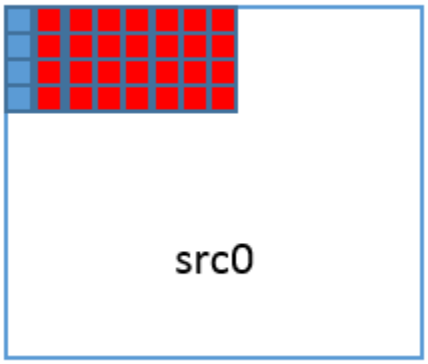
From **cl_intel_subgroups** extension specification page: "*The goal of this extension is to allow programmers to improve the performance of their applications by taking advantage of the fact that some work items in a work group execute together as a group (a "subgroup"), and that work items in a subgroup can take advantage of hardware features that are not available to work items in a work group. Specifically, this extension is designed to allow work items in a subgroup to share data without the use of local memory and work group barriers, and to utilize specialized hardware to load and store blocks of data.*"

The size of subgroup is equal to SIMD width (note that code targeting Intel® Processor Graphics can be compiled SIMD-8, SIMD-16, or SIMD-32 depending on the size of the kernel, which means that 8, 16 or 32 work items respectively can fit on a hardware thread of the Execution Unit (EU). For a deeper introduction please see section 5.3.5 SIMD Code Generation for SPMD Programming Models of Stephen Junkins' excellent paper "The Compute Architecture of Intel® Processor Graphics Gen8"). For example if the kernel is compiled SIMD-8, then a subgroup is made up of 8 work items that share 4 KB of register space of a hardware thread and execute together. The programmers could use a kernel function `get_sub_group_size` to figure out the size of the subgroup.

We mainly use two kernel functions in this sample: `intel_sub_group_shuffle` and `intel_sub_group_block_read`. We use `intel_sub_group_shuffle` to share data between work items in a subgroup; we use `intel_sub_group_block_read` to read a block of data for each work item in a subgroup from a source image at a specific location.

Let's take a look at the code below. Assume the subgroup size is 8. A block read function `intel_sub_group_block_read4` reads four uints of data from a source image in **row-major** order and after conversion to four floats stores the value into a blockA private variable of each work item in a subgroup. The data for the first work item is shown as four blue blocks in the first column (see the diagram below). Then we read the value of private variable blockA of a work item with a sub group local id provided as a second parameter to the `intel_sub_group_shuffle` into variables acol0-7. After performing eight sub group shuffles, each work item has the full data of a 4 by 8 block. . For a detailed explanation of `intel_sub_group_shuffle` and `intel_sub_group_block_read` functions and their operation please refer to **cl_intel_subgroups** extension specification page.

```
float4  blockA = as_float4( intel_sub_group_block_read4( src0, coordA ) );
const float4    acol0 = intel_sub_group_shuffle( blockA, 0 );
const float4    acol1 = intel_sub_group_shuffle( blockA, 1 );
const float4    acol2 = intel_sub_group_shuffle( blockA, 2 );
const float4    acol3 = intel_sub_group_shuffle( blockA, 3 );
const float4    acol4 = intel_sub_group_shuffle( blockA, 4 );
const float4    acol5 = intel_sub_group_shuffle( blockA, 5 );
const float4    acol6 = intel_sub_group_shuffle( blockA, 6 );
const float4    acol7 = intel_sub_group_shuffle( blockA, 7 );
```



OpenCL Implementation

gemm.cl file provided with the sample contains several different implementations that demonstrate how to optimize the SGEMM kernels for Intel® Processor Graphics. We start with a naïve kernel and follow with kernels using local memory and kernels using **cl_intel_subgroups** extension. Since using local memory is a common practice when optimizing an OpenCL kernel, we focus on kernels using **cl_intel_subgroups** extension. At the same time we also use a well-known practice of tiling (or blocking) in these kernels, where matrices are divided into blocks and the blocks are multiplied separately to maintain a better data locality. We tested the performance of our kernels on a 4th Generation Intel® Core™ Processor with Intel® Iris™ Pro Graphics 5200, which contains 40 EUs running at 1.3GHz with a theoretical peak compute[1] of 832 Gflops) and a 5th Generation Intel® Core™ Processor with Intel® Iris™

Graphics, which contains 23 EUs running at 900MHz with a theoretical peak compute of 331 Gflops) running SUSE Linux Enterprise Server* (SLES) 12 GM operating system and Intel® Media Server Studio 16.4.2 release.

The naming convention of the kernels is as follows: ***optimizationMethod_blockHeight x blockWidth_groupHeight x groupWidth***. The matrices in the kernels are in column-major order.

Naïve Kernel

Let's take a look at the naïve implementation first, which is fairly similar to the original C version with just the addition of a few OpenCL C qualifiers, and with the outermost loop replaced by a parallel kernel. The compute efficiency of the naïve kernel is only about 2%~3% in our test environment.

There are two major issues in the naïve code.

1. Global memory is accessed repeatedly without any data reuse/sharing;
2. Each work item only calculates one output without using register space wisely;

```
__kernel void gemm_naive(__global const float * restrict A,
                         __global const float * restrict B,
                         __global float * restrict C,
                         float alpha, float beta, int width0, int width1)
{
    int row = get_global_id(1);
    int col = get_global_id(0);
    float sum = 0;
    for (int i = 0; i < width0; i++)
        sum += A[row*width0+i] * B[i*width1+col];

    C[row*width1+col] = alpha * sum + beta * C[row*width1+col];
}
```

Kernels using Local Memory

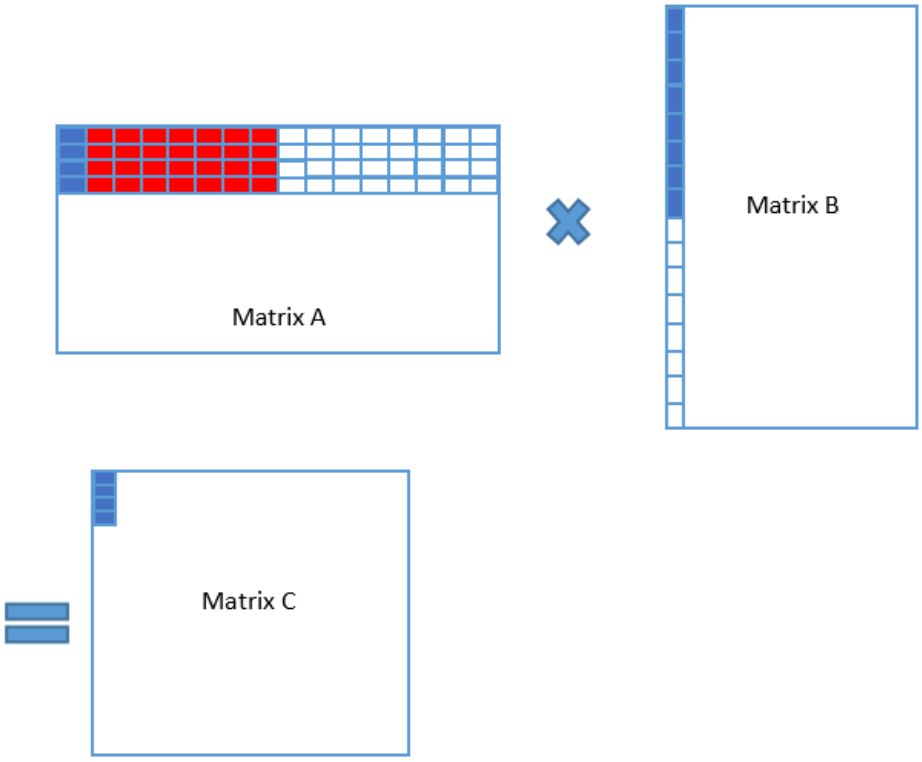
L3_SLM_xx_xx kernels load matrix A into local memory, synchronize work items with a barrier, and then proceed with the computation. Using local memory is a common optimization to avoid repeated global memory access. The compute efficiency of these kernels is about 50% in our test environment.

Kernels using cl_intel_subgroups Extension

We developed two different types of kernel using **cl_intel_subgroups** extension. One is L3_SIMD_xx_xx loading data from a regular OpenCL buffer, and the other one is block_read_xx_xx reading a block of data from OpenCL image2D by using `intel_sub_group_block_read`. Kernels differ due to different ways of input data access, but the basic idea of data sharing in a subgroup is similar. Let's take *block_read_4x1_1x8* kernel as an example. Note that in this example tiling sizes are too small to be efficient and are used for illustration purposes only. According to the above naming convention the kernel handles 4 * 1 floats in a work item and a work group size is (1, 8). The following picture shows

how the kernel works in a subgroup. The partial kernel code is also shown in "**cl_intel_subgroups** Extension" section.

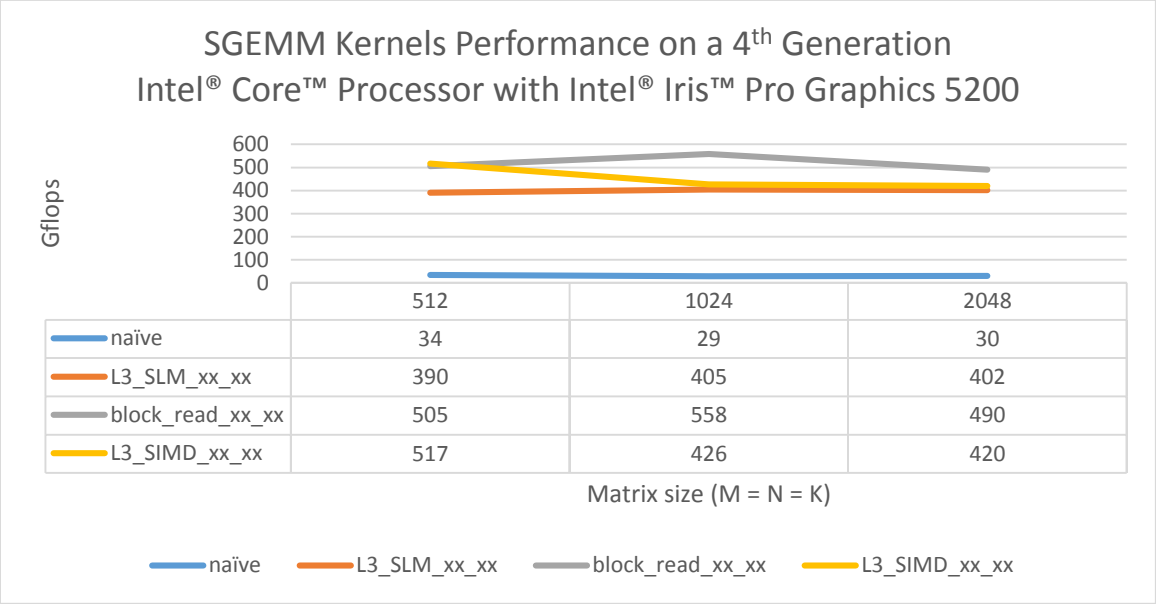
This kernel is compiled to SIMD-8, thus the subgroup size is 8. In matrix A, a float4 is read by `intel_sub_group_block_read4` in a row-major order at first (refer to 4 blue blocks in matrix A), then `intel_sub_group_shuffle` is called to share adjacent 7 columns of float4 from work item 1~7 in the 1st subgroup (refer to 28 red blocks in matrix A). In matrix B, a float8 is read by `intel_sub_group_block_read8` in row-major order as well (refer to 8 blue blocks in matrix B). We need to read a float8 from matrix B because 8 columns of data in matrix A could be got by shuffle function. After that we could do the sub-matrix multiplication (4 * 8) * (8 * 1) and get the partial result of sub-matrix C (4 * 1). This is the 1st read and calculation in the 1st work item.
Then we will move to the next block of (4 * 8) in column-major order from Matrix A and move to the next block of (8 * 1) in row-major order from Matrix B (Refer to white blocks in matrix A and B). In other word in the 1st work item we walk across the 1st 4 rows of matrix A and walk down the 1st column of matrix B and finally get the 1st (4 * 1) block of matrix C (refer to 4 blue blocks in matrix C). In the following work item, we will move to next 4 rows of matrix A or next column of matrix B.
The tiling parameters TILE_M, TILE_K and TILE_N decide the partial matrix sizes of three large matrices in a work group. In a work-group matrix C's size is TILE_M by TILE_N elements, matrix A's size is TILE_M by TILE_K elements and matrix B's size is TILE_K by TILE_N elements. In the current implementation work group size is (1 * 8), the size of a work-item sub-matrix C is TILE_M/1 by TILE_N/8 elements, the size of a sub-matrix A is TILE_M/1 by (TILE_K/8 * 8) elements and the size of a sub-matrix B is TILE_K/1 by TILE_N/8 elements. For a sub-matrix A, we need to multiply by 8 because we share eight columns of float4 from eight work items in a subgroup by using shuffle function. Thus in this kernel TILE_M = 4, TILE_K is 8 and TILE_N is 8.

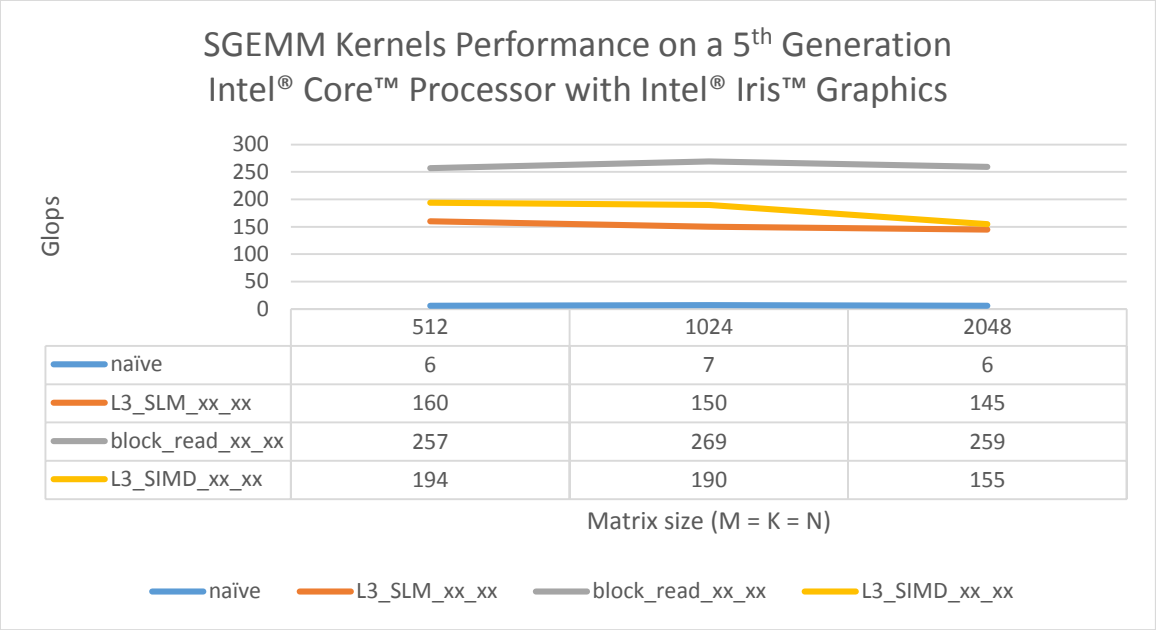


When we apply the **cl_intel_subgroups** extension the kernel performance improves further. The compute efficiency of L3_SIMD_xx_xx is about 60% in our test environment. The compute efficiency of block_read_xx_xx is about 80% on the 5th Generation Intel® Processors, and 65~70% on the 4th Generation Intel® Processors. We will discuss performance of these kernels in the next section.

Performance

Here is the kernel performance comparison between different implementations of SGEMM on a 4th Generation Intel® Core™ Processor with Intel® Iris™ Pro Graphics 5200, which contains 40 EUs running at 1.3GHz, and a 5th Generation Intel® Core™ Processor with Intel® Iris™ Graphics, which contains 23 EUs running at 900MHz, on a SLES 12 GM OS and MSS 16.4.2 release. The block_read_32x2_8x1 and block_read_32x1_8x1 show about 80% compute efficiency on a 5th Generation Intel® Core™ Processor.

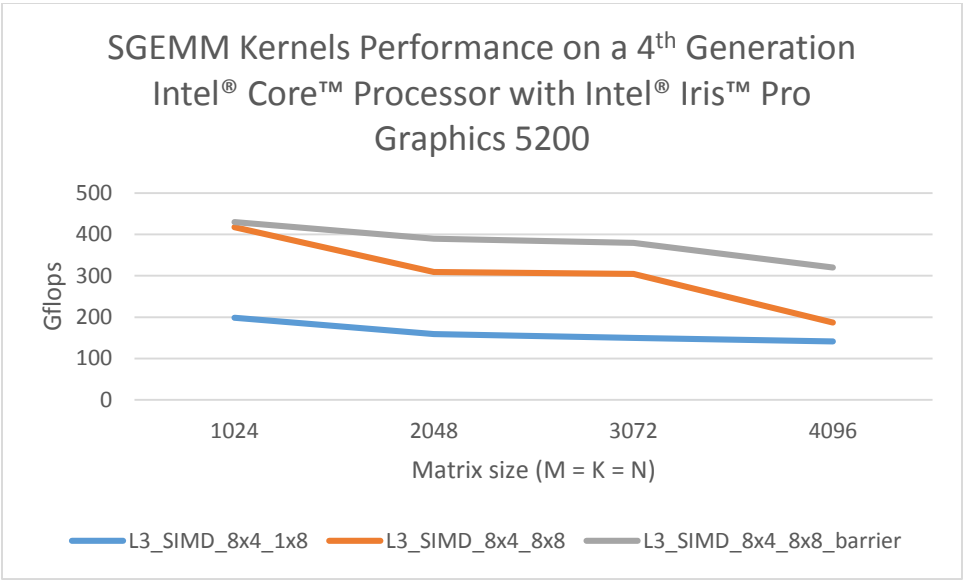


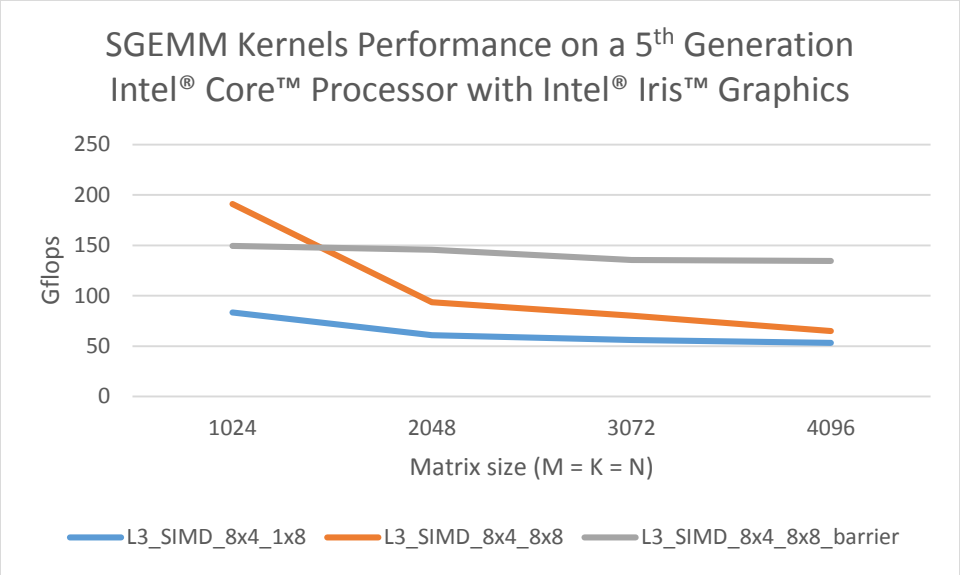


Optimization Tips

Impact of Barriers and Work Group Size on Performance in Non-local Memory Kernels

The built-in function `barrier(CLK_LOCAL_MEM_FENCE)` is commonly used in kernels with local memory, but in non-local memory kernels it may also provide performance benefits when matrix size is too large to fit into cache. Let's take a look at L3_SIMD_8x4_1x8, L3_SIMD_8x4_8x8 and L3_SIMD_8x4_8x8_barrier. L3_SIMD_8x4_1x8 is the basic implementation, and work group size is enlarged from (1 * 8) to (8 * 8) in L3_SIMD_8x4_8x8. L3_SIMD_8x4_8x8_barrier adds a barrier after loading matrix B to make the work items of a work group stay in synch for better L3 cache use. Let's compare the performance when matrix size reaches 1K. The performance improvement can be seen in the following graphs.

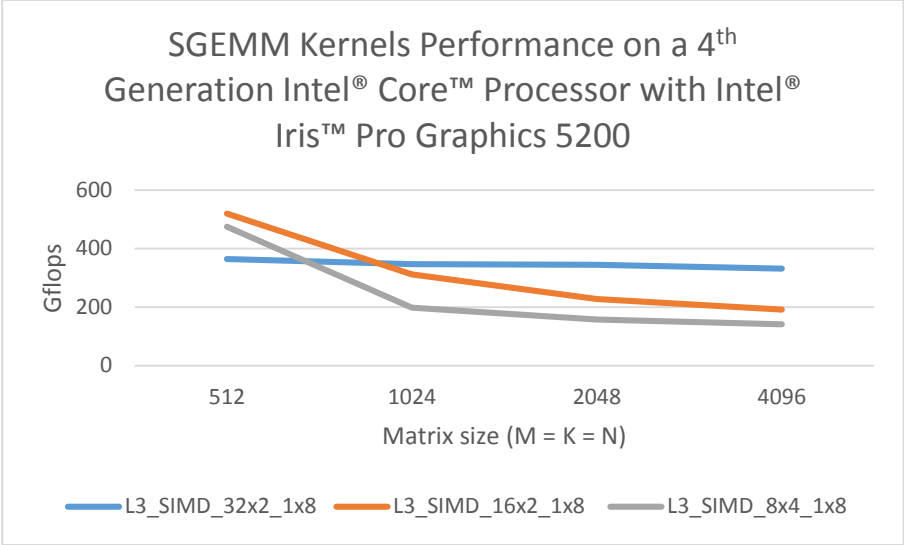


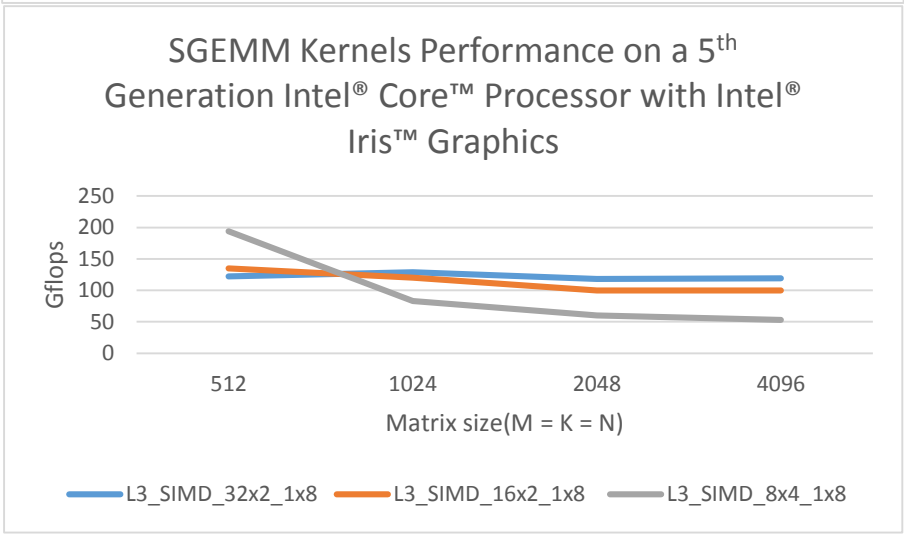


Impact of Tiling Parameters on Performance

The tiling technique generally provides the speedup, but we need to avoid performance degradation due to an overuse of private memory, which exhausts register space. On the 4th and 5th Generation Intel® Core™ Processors each EU thread has 128 general purpose registers. Each register stores 32 bytes, accessible as an 8 element vector of 32 bit data elements, for a total of 4KB. Thus each work item in an OpenCL kernel has access to up to 128, 256, or 512 (SIMD32 / SIMD-16 / SIMD-8) bytes of register space. If the large tiling parameters exceed the private memory threshold, the performance will suffer.

It is hard to find the best tiling parameters that show good performance for different matrix sizes. It is possible that some tiling parameters run faster on some matrix size but slower on others. You may also use some auto-tuning code generator to try other tiling parameters and get the best performance on your device. Another limitation of large tiling parameters is the matrix size must be aligned with large tiling size. Please compare the performance between the kernels like L3_SIMD_32x2_1x8, L3_SIMD_16x2_1x8 and L3_SIMD_8x4_1x8 on different matrix sizes.





The kernels block_read_32x2_1x8 and block_read_32x1_1x8 cannot achieve 80% compute efficiency on Intel® Xeon® Processors E3 with 47 EUs running at 1.15 GHz. We provide block_read kernels with different tiling parameters and work group sizes, like block_read_16x2_1x8, block_read_16x2_4x8, block_read_16x2_4x8 and block_read_16x4_1x8, block_read_16x4_4x8, block_read_16x4_8x8. These kernels perform better than 32x2 and 32x1 kernels, and the best performance can be 80% or close to 80%. Users may need to try different parameters to achieve the best performance on their platforms.

SGEMM Kernels Optimization with Intel® VTune Amplifier XE

We highly recommend to use Intel® VTune Amplifier XE to gain deeper understanding of the application performance on Intel® Processor Graphics. Here we focus on the performance analysis of different optimizations of SGEMM kernel on Intel® Processor Graphics. See the articles by Julia Fedorova in the References section to understand the overall OpenCL performance analysis on Intel® Processor Graphics. In this sample we chose Advanced Hotspots analysis of Intel® VTune Amplifier XE, enabled *Analyze GPU Usage* option, chose *Overview* at *Analyze Processor Graphics events*, enabled OpenCL profiling on the GPU and selected the option *Trace OpenCL kernels on Processor Graphics*.

The testing matrix size is (1024 * 1024) and all the kernels are executed once on Windows 8.1 running on 4th Generation Intel® Core™ Processor with Intel HD4400, which has 20 EUs running at 600Mhz. Here is the VTune screenshot of various SGEMM kernel runs sorted by the execution time:

Grouping: Computing Task Purpose / Computing Task (GPU) / Instance

Computing Task Purpose / Computing Task (GPU) / Insta...	Work Size		Computing Task				Data Transf...		EU Array			GPU Memor...		Sampler		GPU L3 Misses, Misses/sec	Computing Threads Started
	Global	Local	Total Time	Aver...	Inst...	SIM...	Size	Band..	Active	Stall...	Idle	Read	Write	Busy	Bott...		
Compute			606.968ms	50.5 ...	12			0.000	70.0%	29.4%	0.6%	12.668	0.085	0.1%	0.0%	180,976,661.590	100,219
gemm_naive	1024 x 1024	16 x 1	353.298ms	353. ...	1	16		0.000	68.9%	30.9%	0.2%	13.257	0.012	0.0%	0.0%	189,329,681.665	65,411
L3_SIMD_8x4_1x8	256 x 128	8 x 1	40.917ms	40.9 ...	1	8		0.000	46.7%	52.5%	0.8%	15.862	0.100	0.0%	0.0%	206,320,053.401	4,096
L3_SIMD_32x2_1x8	512 x 32	8 x 1	27.930ms	27.9 ...	1	8		0.000	65.6%	34.2%	0.1%	3.552	0.224	1.8%	0.0%	57,666,228.884	2,048
L3_SLM_8x8_16x16	128 x 128	16 x 16	23.235ms	23.2 ...	1	8		0.000	73.5%	25.8%	0.6%	4.221	0.175	0.0%	0.0%	68,042,554.582	2,048
L3_SLM_8x8_4x16	128 x 128	16 x 4	22.943ms	22.9 ...	1	8		0.000	74.2%	24.7%	1.1%	9.137	0.178	0.0%	0.0%	143,672,498.876	2,048
L3_SLM_8x8_8x16	128 x 128	16 x 8	22.054ms	22.0 ...	1	8		0.000	75.4%	21.6%	2.9%	5.932	0.187	0.0%	0.0%	95,165,437.959	2,048
L3_SIMD_8x4_8x8	256 x 128	8 x 8	20.995ms	20.9 ...	1	8		0.000	76.0%	21.4%	2.6%	12.105	0.194	0.0%	0.0%	170,841,917.238	4,088
L3_SIMD_16x2_1x8	512 x 64	8 x 1	20.708ms	20.7 ...	1	8		0.000	79.7%	20.2%	0.1%	19.272	0.197	0.0%	0.0%	243,360,754.581	4,096
L3_SIMD_8x4_8x8_barrier	256 x 128	8 x 8	20.295ms	20.2 ...	1	8		0.000	76.3%	22.3%	1.4%	8.895	0.204	0.0%	0.0%	134,253,973.132	4,096
block_read_32x1_1x8	1024 x 32	8 x 1	20.050ms	20.0 ...	1	8		0.000	74.3%	22.7%	3.0%	22.430	0.207	0.0%	0.0%	351,771,719.491	4,096
L3_SIMD_16x2_4x8	512 x 64	8 x 4	18.515ms	18.5 ...	1	8		0.000	86.1%	13.8%	0.1%	16.865	0.223	0.0%	0.0%	216,023,513.483	4,096
block_read_32x2_1x8	512 x 32	8 x 1	16.028ms	16.0 ...	1	8		0.000	91.4%	7.3%	1.4%	13.928	0.247	0.0%	0.0%	219,924,062.650	2,048

Let's take gemm_naive as an example. At first take a look at the number of compute threads started (hardware threads), which is ~65536, and the formula is Global_size / SIMD_width. Secondly the EU Array stall rate is highlighted in red with a large number of L3 misses, which means EUs are waiting ~30% of the time for data from memory. From this information we could infer that 20% of 65536 of software threads are not doing productive work during kernel execution. Due to a large number of software threads and high EU stall rate, the kernel performs poorly.

Since SGEMM should not be memory bandwidth-bound, we will try to optimize memory accesses and layout. We use common optimization techniques like coalescing memory accesses and utilizing shared local memory (SLM). **cl_intel_subgroups** extension provides another avenue for the optimization. The basic idea is to share the data and let each work item do more work, then the ratio between loading data and computation is more balanced. At the same time using vector data types and block reads also improves the memory access efficiency.

As you can see from the table above, the kernel block_read_32x2_1x8 has the best performance. The EU Array stalls are only 7.3% with 2048 software threads launched. Although each work item takes some time to calculate a block of 32*2 floats, it is likely to hide the memory read stall. The block read and shuffle functions provide efficient memory access, at the same time tiling size of the kernel won't exhaust the register space. We could also compare the data between L3_SIMD_8x4_1x8, L3_SIMD_8x4_8x8 and L3_SIMD_8x4_8x8_barrier. The optimization mentioned in the 1st tip provides better cache performance. L3_SIMD_8x4_8x8 and its barrier version get the benefits from fewer L3 misses and lower rate of EU Array stalls due to the synchronization in the work group.

Controlling the Sample

Option	Description
-h, --help	Show this help text and exit.
-p, --platform <number-or-string>	Selects the platform, the devices of which are used. (Default value: Intel)
-t, --type all | cpu | gpu | acc | default | <OpenCL constant for device type>	Selects the device by type on which the OpenCL kernel is executed. (Default value: gpu)

-d, --device <number-or-string>	Selects the device on which all stuff is executed. (Default value: 0)
-M, --size1 <integer>	Rows of 1st matrix in elements. (Default value: 0)
-K, --size2 <integer>	Cols of 1st matrix in elements, rows of 2nd matrix in element (Default value: 0)
-N, --size3 <integer>	Cols of 2nd matrix in elements (Default value: 0)
-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)
-k --kernel naive | L3_SIMD_32x2_1x8 | L3_SIMD_16x2_1x8 | L3_SIMD_16x2_4x8 | L3_SIMD_8x4_1x8 | L3_SIMD_8x4_8x8 | L3_SIMD_8x4_8x8_barrier | L3_SLM_8x8_8x16 | L3_SLM_8x8_4x16 | L3_SLM_8x8_16x16 | block_read_32x2_1x8 | block_read_32x1_1x8	Determines format of matrices involved in multiplication. There are several supported kernels with naive implementation and optimization on Intel GPU; both matrices A and B are in column-major form; (Default value: NULL)
-v --validation	Enables validation procedure on host (slow for big matrices). (Default disabled)

1. Peak compute is different for each product SKU and is calculated as follows: (MUL + ADD) x Physical SIMD x Num FPUs x Num EUs x Clock Speed, where Physical SIMD is 4 for Intel® Processors with Intel® Processor Graphics.

Conclusion

In this article we demonstrated how to optimize Single precision floating General Matrix Multiply (SGEMM) algorithm for the best performance on Intel® Core™ Processors with Intel® Processor Graphics. We implemented our sample using OpenCL and relied heavily on Intel's **cl_intel_subgroups** OpenCL extension. When used properly, **cl_intel_subgroups** OpenCL extension provides an excellent performance boost to SGEMM kernels.

References

1. Wikipedia page on Matrix Multiplication
2. BLAS Wikipedia page
3. "The Compute Architecture of Intel® Processor Graphics Gen8" by Stephen Junkins
4. **cl_intel_subgroups** extension specification page
5. "Intel® VTune™ Amplifier XE: Getting started with OpenCL* performance analysis on Intel® HD Graphics" by Julia Fedorova
6. "Analyzing OpenCL applications with Intel® VTune™ Amplifier XE" Webinar by Julia Fedorova (please use Internet Explorer to view the videos).
7. Intel® VTune™ Amplifier 2015
8. Optimizing Simple OpenCL Kernels: Modulate Kernel Optimization by Robert Ioffe
9. Optimizing Simple OpenCL Kernels: Sobel Kernel Optimization by Robert Ioffe

About the Authors

Lingyi Kong is a Software Engineer at Intel's IT Flex Services Group. He is an expert in GPU programming and optimization, and also has Graphics driver/runtime development experience on Intel® Iris and Intel® Iris Pro Graphics.

Robert Ioffe is a Technical Consulting Engineer at Intel's Software and Solutions Group. He is an expert in OpenCL programming and OpenCL workload optimization on Intel Iris and Intel Iris Pro Graphics with deep knowledge of Intel Graphics Hardware. He was heavily involved in Khronos standards work, focusing on prototyping the latest features and making sure they can run well on Intel architecture. Most recently he has been working on prototyping Nested Parallelism (enqueue_kernel functions) feature of OpenCL 2.0 and wrote a number of samples that demonstrate Nested Parallelism functionality, including GPU-Quicksort for OpenCL 2.0. He also recorded and released two Optimizing Simple OpenCL Kernels videos and a third video on Nested Parallelism.

Download Code