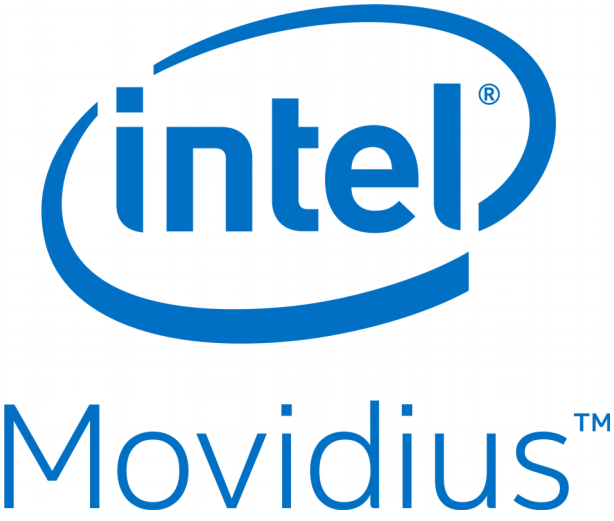


BLIS Test App for Myriad 2

Application Note

v1.01 / February 2018

Copyright and Proprietary Information Notice

Intel Movidius
2200 Mission College Blvd
M/S SC11-201
Santa Clara, CA 95054
http://www.movidius.com/

Revision History

Date	Version	Description
February 2018	1.01	Logomark, legal and cosmetic updates. Updated BLIS code links to point to https://github.com/flame/blis (formerly https://code.google.com/p/blis/).

Table of Contents

1 Introduction

This document provides information on the GEMM and TRSM BLIS kernels implementation for the Myriad 2 architecture.

The development started on top of the BLIS reference code, version 0.1.0-34. The latest version the BLIS source code can be downloaded using "`git clone https://github.com/flame/blis`". The main idea was to use the BLIS micro-kernels (GEMM, TRSM and GEMMTRSM), previously developed by Codec-Art for Movidius, and schedule each "micro-operation" on an individual SHAVE processor. A micro-operation is a single micro-kernel execution involving submatrices of A, B and C, called micro-panels.

Any BLIS optimized implementation has at least two steps. The first step is the development of optimized micro-kernels specific to the instruction set of the target architecture, while the second consists in finding the optimal strategy for scheduling the micro-kernel execution. This step has to do more with memory constraints, communication between different execution units, DMA and others. The first delivery contained a BLIS version running on Leon, with micro-kernel code written in SHAVE assembly language and only a basic implementation of micro-kernel scheduling, following the reference algorithm, which was basically the first step in the BLIS optimized implementation. The next step, the BLISTestApp, described in this document deals with finding the optimal scheduling strategy.

- **Glossary**

BLIS	BLAS-like Library Instantiation Software
GEMM	GEneral Matrix Multiply
TRSM	TRiangular Solver with Multiple right hand terms
FRAGRAK	Myriad 2

2 Background

In the initial LAMATestApp, the packing, unpacking and scheduling of micro-operation was done by Leon and only one SHAVE executed a single micro-operation. The data transfer from Leon to SHAVE was performed via DDR and distributing micro-operations across multiple SHVAEs was useless, as they were all executed serially.

In BLISTestApp, we grouped multiple micro-operations, up to 16, into a batch and executed the entire batch on a single SHAVE processor. Also, we have separated independent micro-operations and executed them on different SHAVEs, in order to allow parallel executions. Another feature added to the original BLIS code was the DMA transfer of data from DDR to CMX, to allow for less memory access stall cycles for the micro-kernel code.

3 Kernels

3.1 CMX buffers

There are five delivered kernels: GEMM and four TRSM flavors (LL – left lower, LU – left upper, RL – right lower and RU – right upper). All the kernels require data matrix transfer from DDR to CMX. In order to make this transfer more flexible, CMX buffers were statically allocated for each SHAVE, for each of A, B and C matrices.

For matrix A, for each SHAVE, a chunk of 64 kB is reserved in its corresponding CMX slice, memory space reserved for B is 4 kB and for C matrix only 1 kB. This space is enough to hold 16 micro-panels of 4 x 128 (maximum value for k) float values.

3.2 SHAVE processors

There is a macro that controls the maximum number of SHAVEs used by the BLIS kernels. The number of SHAVEs used in parallel is controlled by the global variable "numberShaves".

- MAX_SHAVES – always 12 for Myriad 2, the number of SHAVE processors present on the platform.
- numberShaves – which is the number of SHAVE processor that are actually going to be used.

At the moment, users of the BLIS functions can decide at compile time what is the maximum number of SHAVE processors to use. However, no matter how many SHAVE processors will be used, the workload one processor can do at once is the same, that is the maximum number of micro-operations the SHAVE can process at a single call from Leon, 16. This number is limited by the current memory configuration, but a scenario in which the workload is adjusted to the number of SHAVEs used can be envisioned, in order to reduce the Leon/SHAVE switch.

3.3 GEMM

The GEneral Matrix Multiply kernel performs the matrix operation:

C := alpha*A*B + beta*C,

where A is an `m X n` matrix, B is an `n X p` matrix and C is an `m X p` matrix. Since the matrix multiplication A*B can be rewritten in terms of block matrices, submatrices of A and B. The dimension of this submatrices, called micro-panels, are 4 X k for A and k X 4 for B, with k having a maximum value of 128. The multiplication of two micro-panels is a micro-operation and it is executed in a single GEMM micro-kernel call. In the case of GEMM, all the micro-operations are independent, meaning that no micro-operation needs data from the output of another micro-operation, thus the scheduling is more flexible.

We kept the same matrix stepping scheme, with an outer loop going over the columns of C and an inner loop going over the rows of C. In the reference code, at any pointer in any of the two directions there is the starting address of a micro-panel. In our code, each row of micro-panels, which by default contains 16 micro-panels, is grouped into a batch and passed to a SHAVE processor for execution. The next row batch is passed to the next available SHAVE processor and so forth.

By default, BLIS limits the row dimension to 16 micro-panels and, for larger matrices, it splits the original GEMM problem into separate sub-problems, each one solved by a kernel call. Thus, the CMX buffer dimension choices are optimal for this kernel. Since the micro-operations are independent, the best performance would be achieved if the all the available SHAVE processor would perform them in parallel.

However, this is not possible in the current set-up, due to the need of bringing data from DDR to CMX and to store the C matrix back to its original location – DDR. If CMX memory would have been large enough to hold A, B and C, the best performance would have been reached, with only an overhead for copying the A, B and C data to CMX, at the beginning, and for writing C back to DDR.

Thus, some data management techniques were employed in order to best make use of the limited CMX memory, to keep the SHAVE processors waiting for data as little as possible and to prevent race conditions. Some of these techniques were: double buffering the B micro-panels, triple-buffering the C micro-panels, in order to allow one batch of micro-panels to be processes, the next batch to be prepared (copied to CMX) and the previous batch to be written back to DDR. Another technique was to reduce the waiting time for the DMA transfers by grouping them.

3.4 TRSM

The TRiangular Solver with Multiple right hand terms kernel performs one of the matrix operations:

C := alpha*inv(A)*B – left version

C := alpha*B*inv(A) – right version

where A is a m X m triangular matrix, B is a m X n matrix and C is a m X n matrix. For each of the left or right versions there are two subversions, lower and upper, depending on whether matrix A is lower or upper triangular.

Prior to calling the TRSM kernels, BLIS does a diagonal block inversion, meaning that the diagonal values of A are replaced with their inverses, in order to avoid repeated time consuming divisions inside the micro-kernel, thus transforming the triangular solving into a matrix multiplication problem. With the difference that the operations are no longer independent row wise. There is a bottom-up dependency, for upper A, and a top-down dependency, for lower A, just as in a triangular solver. This dependencies impose a particular order in which the micro-operations can be executed.

Just as in GEMM case, the TRSM kernels work on micro-panels and, depending on whether the A micro-panel intersects the diagonal or not, one of GEMMTRSM or GEMM micro-kernels is called for that particular micro-operation.

Also, besides packing the matrices into micro-panels, BLIS does also a problem split, limiting the number of micro-panels on a row to 16 and calling the TRSM kernel for each of the separate problems. The TRSM micro-panels fit in the same buffers that were used for GEMM, thus we can reuse the buffer interface.

The main differences between the GEMM and TRSM kernels are:

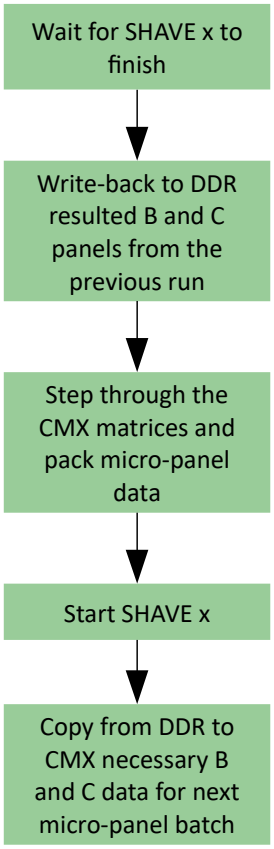
- TRSM kernel calls both of GEMM and GEMMTRSM micro-kernels, while GEMM kernel calls only GEMM micro-kernel.
- the micro-operation execution order, for TRSM, is constrained by the result dependency mentioned earlier.
- the is an update of the B matrix with the result of the previous execution, for TRSM, which implies a second DDR write-back, besides C matrix.

In the case of TRSM LL, there is a top-down order of the micro-operations. Meaning that the micro-operation on the second row micro-panel takes as input the result from the first row micro-operation. As a consequences, it becomes straight-forward to group the execution of dependent micro-operations on the same SHAVE processor, in order to avoid the need of SHAVEs waiting for each other and to be able to have the SHAVEs running in parallel. For this reason, each iteration over the columns of C is associated with a particular SHAVE processor, which gets a 16 micro-panel batch for each of A, B and C matrices to process.

When the last SHAVE processor available is designated for micro-operation execution, the pointer is reset to SHAVE 0. However, since SHAVE 0 was previously assigned a batch of micro-operations to process, Leon must first wait for SHAVE 0 to finish them and, as soon as that happens, write the result data of B and C matrices from CMX to DDR via DMA. Also, upon starting a SHAVE processor, Leon has to copy matrix data from DDR to CMX, in order to fill the CMX buffers with next micro-panel batch for the next run of that particular SHAVE.

For TRSM RL, the operation is a bit simpler, because the matrix B is no longer modified by the micro-kernel, so there is no need for a DDR write-back. This fact will be visible in the performance graphs, where it can be seen that the right hand versions of the TRSM kernel have better figures than left hand ones.

The loop diagram looks like:



There could be variations for this diagram, depending on the particular matrix arrangement, but the general approach is to let Leon processor schedule the SHAVE processors to do the matrix operations, while managing the data back and forth in the background.

For optimal performance, there is a kernel for each of the equation system layout, right or left, and for each triangular matrix arrangement, lower or upper, although only two kernels would have been necessary, LL and LU. An RU kernel can be converted into an LL kernel by transposition. Same for RL and LU. In fact, inside the TRSM code, before calling the kernel, such conversions are made for optimal access to C matrix, depending on whether C is row stored (column stride of one element) or column stored (row stride of one element).

4 Code

The code structure follows the same layout as for BLISTestApp. The only modifications were made in the GEMM and TRSM kernels and in the SHAVE micro-kernel interface.

The configuration specific to Myriad is described by some switches guarded by Myriad 2 flags defined in Makefile depending of MV_SOC_PLATFORM. Also, new directories were created specific to Myriad.

BLIS specific directories

The same configuration as BLIS reference code:

```
leon\config\reference\*
leon\frame\*\*\*
leon\testsuite\src\*
```

4.1 Myriad specific directories

`\config`	Contains custom.ldscript file.
`\doc`	Application document description and performance excel file.
`\input`	input.general and input.operations files that contains the input parameter configuration for testsuite.
`\output`	Output files resulted from building process.
`\scripts`	Scripts file used for debugger and simulation.
`\shared`	Files used for both LEON and SHAVE compilation.
`\shave`	Each entry for every micro-kernel and asm versions of the micro-kernels.

4.2 Tests

We used the same "TESSUITE" test plan that we have used for BLISTestApp. For this reason, only "TESTSUITE" tests were kept, in order to minimize the code generated. "TEST" test plan, that uses the BLIS object interface was removed.

The "TESTSUITE" tests are test cases that were designed to measure the output error, with respect to the reference output, the performance gain, with respect to the reference C code, and the performance scalability.

The BLIS "TESTSUITE" test plan reads parameters from two input files that are loaded from /input directory in the memory: **input.general** and **input.operations** in order to choose which tests to run and how those tests are run.

The **input.general** input file contains parameters that control the general behavior of the test plan.

The **input.operations** input file determines which operations are tested, which parameter combinations are tested, and the relative sizes of the operation's dimensions.

The test parameters, as requested by Movidius, for **input.general** are the following:

1	# Number of repeats per experiment (where best result is reported).
r	#cg # Matrix storage scheme(s) to test ('c' = col-major; 'r' = row-major; 'g' = general stride).
R	#cji # Vector storage scheme(s) to test ('c' = colvec/unit; 'r' = rowvec/unit; 'j' = colvec/non-unit; 'i' = rowvec/non-unit).
0	# Test all combinations of storage schemes?
32	# General stride spacing (for cases when testing general stride).
s	#sdcz # Datatype(s) to test.
64	# Problem size: first to test.
1024	# Problem size: maximum to test.
64	# Problem size: increment between experiments.
1	# Error-checking level (0 = disable error checking; 1 = full error checking).
i	# Reaction to test failure ('i' = ignore; 's' = sleep() and continue; 'a' = abort)z.
1	# Output results in Matlab/Octave format (0 = output without formatting).
0	# Output results to stdout AND files (0 = output only to stdout).

The output to the test suite has the same format as for BLIS reference "TESTSUITE" and can be interpreted identically. See https://github.com/flame/blis/wiki/Testsuite.

5 Compiling

```
make -j4 MV_SOC_PLATFORM=myriad2
```

6 Running

There are 2 ways to run the project: the first one is OS independent and the second one is for Windows only.

6.1 OS independent

6.1.1 For running default test:

- in a Cygwin terminal type:

```
make start_server
```

- in a second Cygwin terminal type:

```
make run
```

6.1.2 For running specific tests:

- in a Cygwin terminal type:

```
make start_server
```

- in a second Cygwin terminal type:

```
make load SourceDebugScript=scripts/test.scr
```

Please replace "test.scr" with one of the following testcases available:

1. BLISTestAppTestSuite_1SHAVE.scr : run TestSuite using one SHAVE.
2. BLISTestAppTestSuite_2SHAVEs.scr : run TestSuite using two SHAVEs.
3. BLISTestAppTestSuite_4SHAVEs.scr : run TestSuite using four SHAVEs.
4. BLISTestAppTestSuite_8SHAVEs.scr : run TestSuite using eight SHAVEs.

Results will be displayed on the "make run" terminal:

6.2 Windows only

Double click on *.bat

The following testcases are available:

1. BLISTestAppTestSuite_1SHAVE.bat : run TestSuite using one SHAVE.
2. BLISTestAppTestSuite_2SHAVEs.bat : run TestSuite using two SHAVEs.
3. BLISTestAppTestSuite_4SHAVEs.bat : run TestSuite using four SHAVEs.
4. BLISTestAppTestSuite_8SHAVEs.bat : run TestSuite using eight SHAVEs.

Results will be displayed on the terminal opened by the .bat file.

7 Performance

The test plan had three objectives:

1. Compare the output of the kernels to that of the reference code;
2. Verify the performance gain over the reference code running on Leon;
3. Observe the scalability of the code over multiple SHAVEs.

During a test, the input matrix A is transposed, conjugated, or made unit triangular, but we will only show below result for the operations with the matrix as it is generated, with no primer operation applied to it (n option). Also, all the matrices will be row stored (r option), as it was requested in the test specifications. This options are coded in the name of the test. For example, bli_strsmll_nn_rrr, means a TRSM Left Lower (trsmll), with matrix A not being transposed or conjugated, and A not being made unit triangular (trsmll_nn), and all the matrices A, B and C being row stored (trsmll_nn_rrr), with single precision floating point values (strsm_ll_rrr).

Nevertheless, all the test have been performed, with all the options for A activated, and the results can be consulted in the spreadsheet attached to this document.

7.1 GEMM

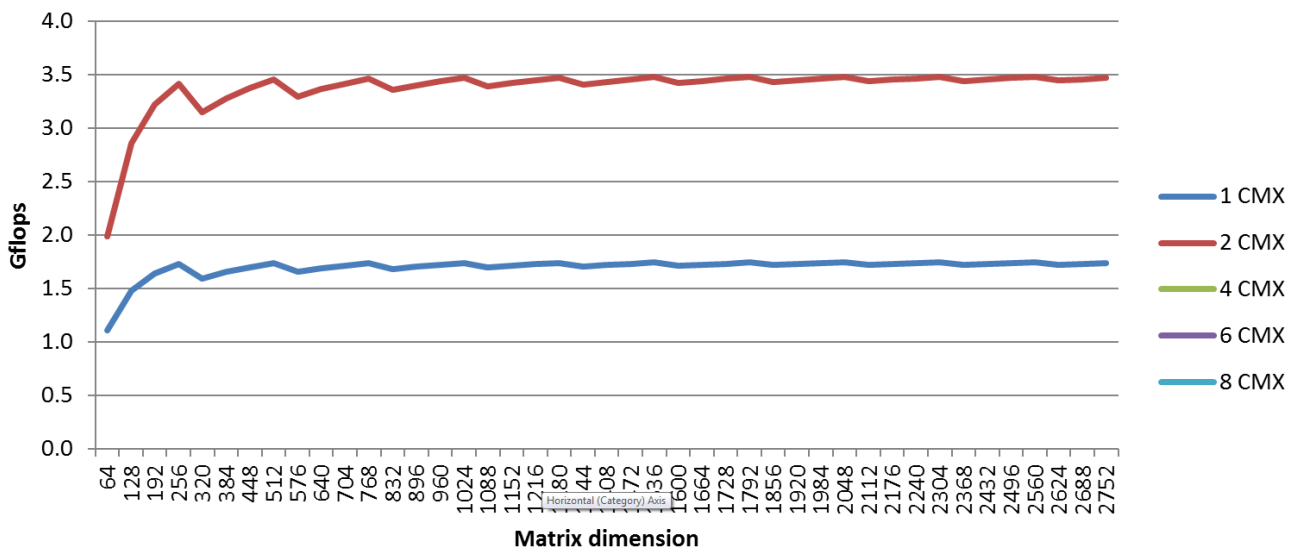


Figure 1: GEMM performance graph

As it can be seen in the above figures, there is an approximate seven fold increase in performance when moving the matrix data from DDR to CMX. Also, there is a good scalability up to 8 SHAVEs, meaning that the micro-operations are well scheduled and the SHAVEs don't have to wait for Leon to do its processing. Another noticeable effect is the zigzag in the performance graph, especially for 8 running SHAVEs.

7.2 TRSM Left Lower (LL)

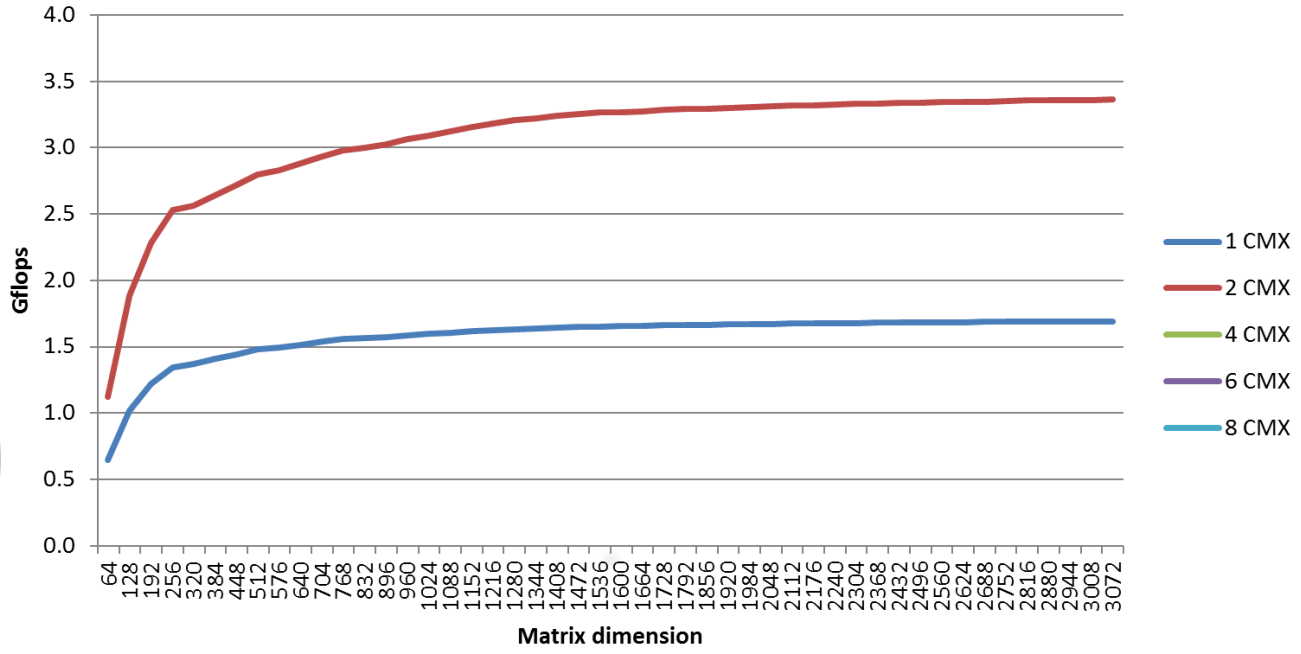


Figure 2: TRSM LL performance graph

7.3 TRSM Left Upper (LU)

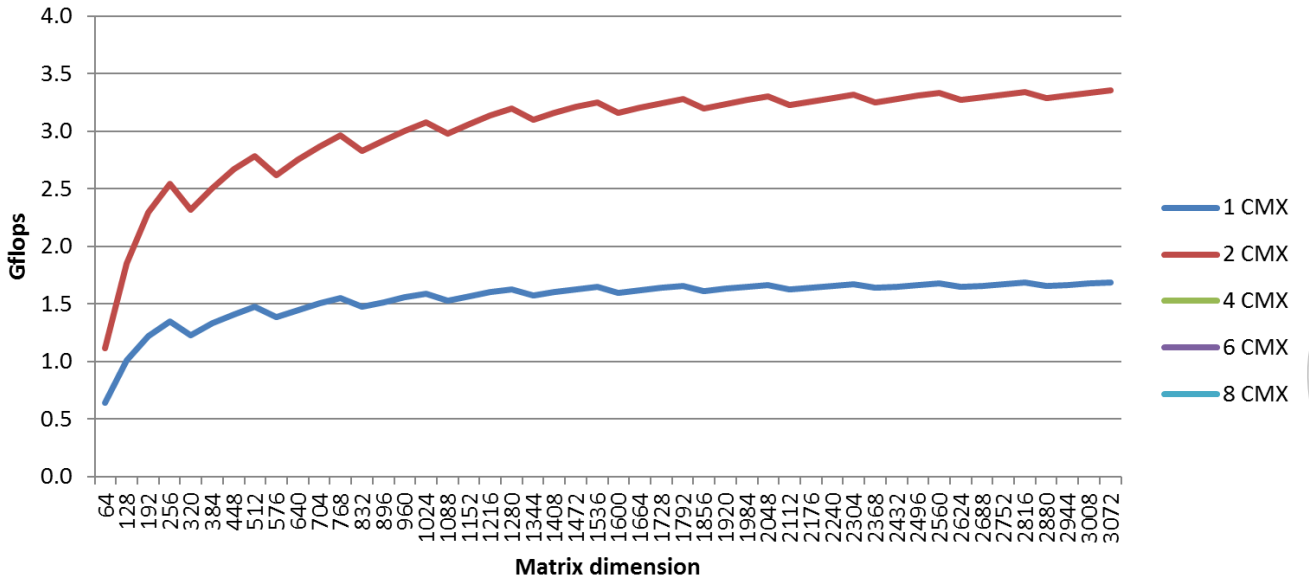


Figure 3: TRSM LU performance graph

7.4 TRSM Right Lower (RL)

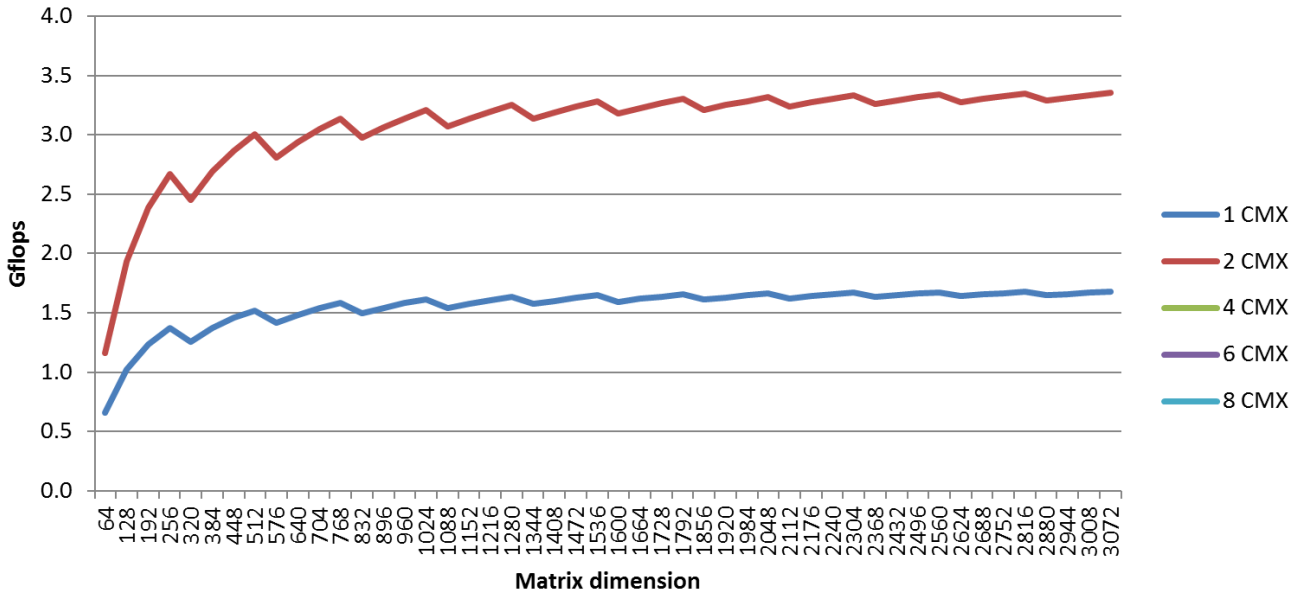


Figure 4: TRSM RL performance graph

7.5 TRSM Right Upper (RU)

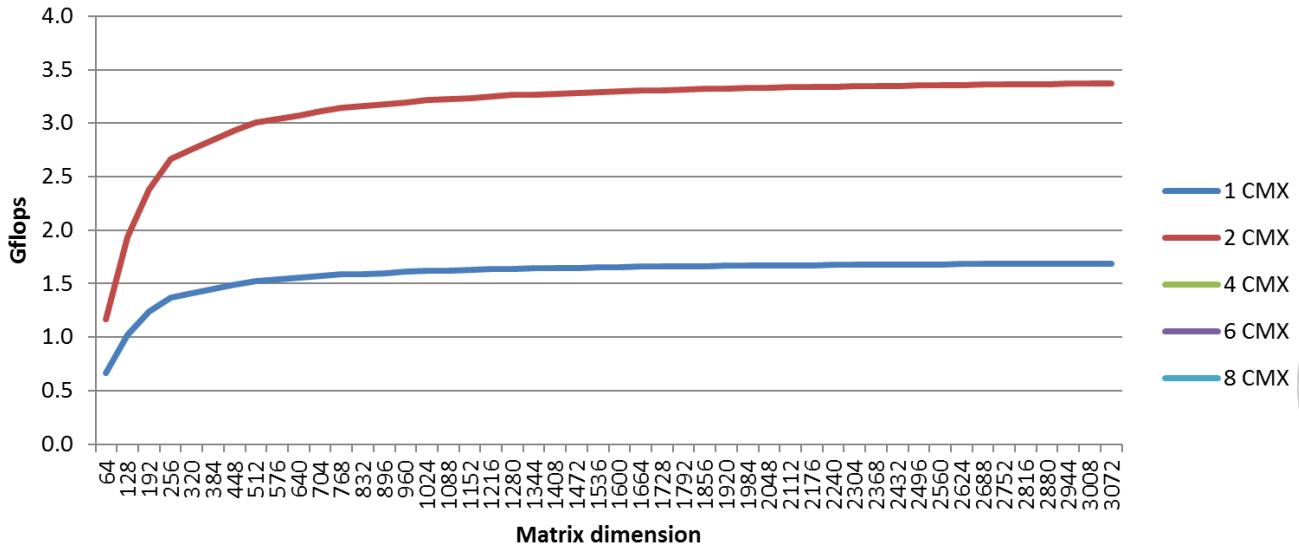


Figure 5: TRSM RU performance graph

8 Myriad model

The BLIS Myriad performance model was constructed to be used for performance estimation of standard BLIS TRSM test cases running over different Myriad configurations. It has a set of configurable parameters (DMA transfers speed, SHAVE frequency, number of SHAVEs used and others) and takes the size of the matrices as inputs. The output is the performance figure expressed in Gflops.

The underlying model requires some key values to be computed from the matrix dimensions, the BLIS configuration parameters and the TRSM kernel specific figures. These values are:

- number of bytes transferred from DDR to CMX for A matrix

$$A_{bytes} = \frac{1}{2} * sizeof(float) * MR^2 * \left\lceil \frac{m}{MR} \right\rceil * \left(\left\lceil \frac{m}{MR} \right\rceil + 1 \right)$$

- number of bytes transferred from DDR to CMX for B matrix

$$B_{bytes} = sizeof(float) * NR * \left\lceil \frac{n}{NR} \right\rceil * \left(k_{MAX} * \left\lceil \frac{k}{k_{MAX}} \right\rceil * \left\lceil \frac{m}{m_{MAX}} \right\rceil + MR * \left\lceil \frac{k \,\%\, k_{MAX}}{MR} \right\rceil * \left\lceil \frac{m - \left\lceil \frac{k}{k_{MAX}} \right\rceil * k_{MAX}}{m_{MAX}} \right\rceil \right)$$

- number of bytes transferred from DDR to CMX (and back) for C matrix.

$$C_{bytes} = sizeof(float) * MR * NR * \left\lceil \frac{n}{NR} \right\rceil * \left(\left\lceil \frac{k}{k_{MAX}} \right\rceil * \left\lceil \frac{m}{MR} \right\rceil - \frac{1}{2} * \left\lceil \frac{k_{MAX}}{MR} \right\rceil * \left\lceil \frac{k}{k_{MAX}} \right\rceil * \left(\left\lceil \frac{k}{k_{MAX}} \right\rceil - 1 \right) \right)$$

- number of GEMMTRSM kernel calls with k=0

$$GEMMTRSM_{calls\,0} = \left\lceil \frac{k}{k_{MAX}} \right\rceil * \left\lceil \frac{n}{NR} \right\rceil$$

- number of GEMMTRSM calls with k not 0

$$GEMMTRSM_{calls\,1} = \left\lceil \frac{n}{NR} \right\rceil * \left(\left(\left\lceil \frac{k_{MAX}}{MR} \right\rceil - 1 \right) * \left\lceil \frac{k}{k_{MAX}} \right\rceil + \left(\left\lceil \frac{k \,\%\, k_{MAX}}{MR} \right\rceil - 1 \right) \right)$$

- sum of all k's for GEMMTRSM calls

$$GEMMTRSM_k = 2 * \left\lceil \frac{n}{NR} \right\rceil * \left(\left(\frac{k_{MAX}}{MR} - 1 \right) * \left(\frac{k_{MAX}}{MR} \right) * \left\lceil \frac{k}{k_{MAX}} \right\rceil + \left(\left\lceil \frac{k \,\%\, k_{MAX}}{MR} \right\rceil - 1 \right) * \left(\left\lceil \frac{k \,\%\, k_{MAX}}{MR} \right\rceil \right) \right)$$

- number of GEMM calls with k equal to k_MAX (256)

$$GEMM_{callsMAX} = \left\lceil \frac{n}{NR} \right\rceil * \left(\left\lceil \frac{k}{k_{MAX}} \right\rceil * \left\lceil \frac{m}{MR} \right\rceil - \frac{1}{2} * \left\lceil \frac{k_{MAX}}{MR} \right\rceil * \left\lceil \frac{k}{k_{MAX}} \right\rceil * \left(\left\lceil \frac{k}{k_{MAX}} \right\rceil + 1 \right) \right)$$

- sum of all k's for GEMM calls

$$GEMM_k = GEMM_{callsMAX} * k_{MAX}(256)$$

➢ **Myriad architecture specific parameters:**

F_{DC} = 266 MB/s – DMA speed for DDR to CMX transfer.

F_{CD} = 907 MB/s – DMA speed for CMX to DDR transfer.

F_{SHAVE} = 180 MHz – frequency of a SHAVE processor.

N_{DMA} = 4 – number of DMA tasks running in parallel.

N_{SHAVE} = 1 – number of SHAVEs running in parallel.

➢ **BLIS kernel specific parameters:**

MR = 4 – number of column elements for A and C micro-panels.

NR = 4 – number of row elements for B and C micro-panels.

m_{MAX} = 64 – maximum number of column elements of A passed to a kernel call.

k_{MAX} = 256 – maximum number of row elements of A passed to a kernel call.

$Cycles_{out0}$ = 52 – cycle count for GEMMTRSM with k equal to 0.

$Cycles_{out1}$ = 73 – cycle count for GEMMTRSM with k not 0.

$Cycles_{loop}$ = 20 – cycle count for GEMMTRSM and GEMM loop around k.

$Cycles_{out}$ = 75 – cycle count for GEMM with k not 0.

These parameters are needed in order to expose the architecture to the model, thus allowing the model to be customized for a new architecture. Also, key numbers of the micro-kernel s are provided, because they help describe the flow of data. These figures are combined in the above equations to give the deterministic performance figure. However, this figure is not enough to explain all the code interactions, which are harder to model analytically, and for these we employed a polynomial fitting approach. Three different polynomials were obtained, for a better fitting, depending on the input matrix size. The coefficients are presented below.

➢ **Fitted coefficients using Octave polyfit:**

for m < 512: $C_3 = 1.837E^{-10}$, $C_2 = 7.3157E^{-8}$, $C_1 = -6.3061E^{-6}$, $C_0 = 5.0709E^{-4}$

for 512 ≤ m < 768: $C_3 = 3.6489E^{-10}$, $C_2 = -2.6508E^{-7}$, $C_1 = 2.1246E^{-4}$, $C_0 = -4.9326E^{-2}$

for 768 ≤ m: $C_3 = 4.9532E^{-10}$, $C_2 = -7.4652E^{-7}$, $C_1 = 7.4028E^{-4}$, $C_0 = -2.3641E^{-1}$

➢ **Execution time model:**

$$Exec\_time = (((A_{bytes}+B_{bytes}+C_{bytes})/F_{DC}+C_{bytes}/F_{CD})/N_{DMA}+(GEMMTRSM_{calls0}*Cycles_{out0}+ GEMMTRSM_{calls1}*Cycles_{out1}+ GEMMTRSM_{k}*Cycles_{loop}/4+GEMM_{callsMAX}*Cycles_{out}+GEM_{k}*Cycles_{loop}/4)/F_{SHAVE})/N_{SHAVE}/ ScalingFactor+m^3*C_3+m^2*C_2 + m*C_1+ C_0$$

ScalingFactor, between 0 and 1, was added to the model because the performance doesn't scale exactly when adding another SHAVE for processing. Also, there is a different polynomial pair, which fits the execution time difference, for each SHAVE number combination. All the polynomial coefficients, together with the scaling factors, are given in the attached spreadsheet.

➢ **Performance model:**

Gflops = m^3 / Exec_time / 10^9

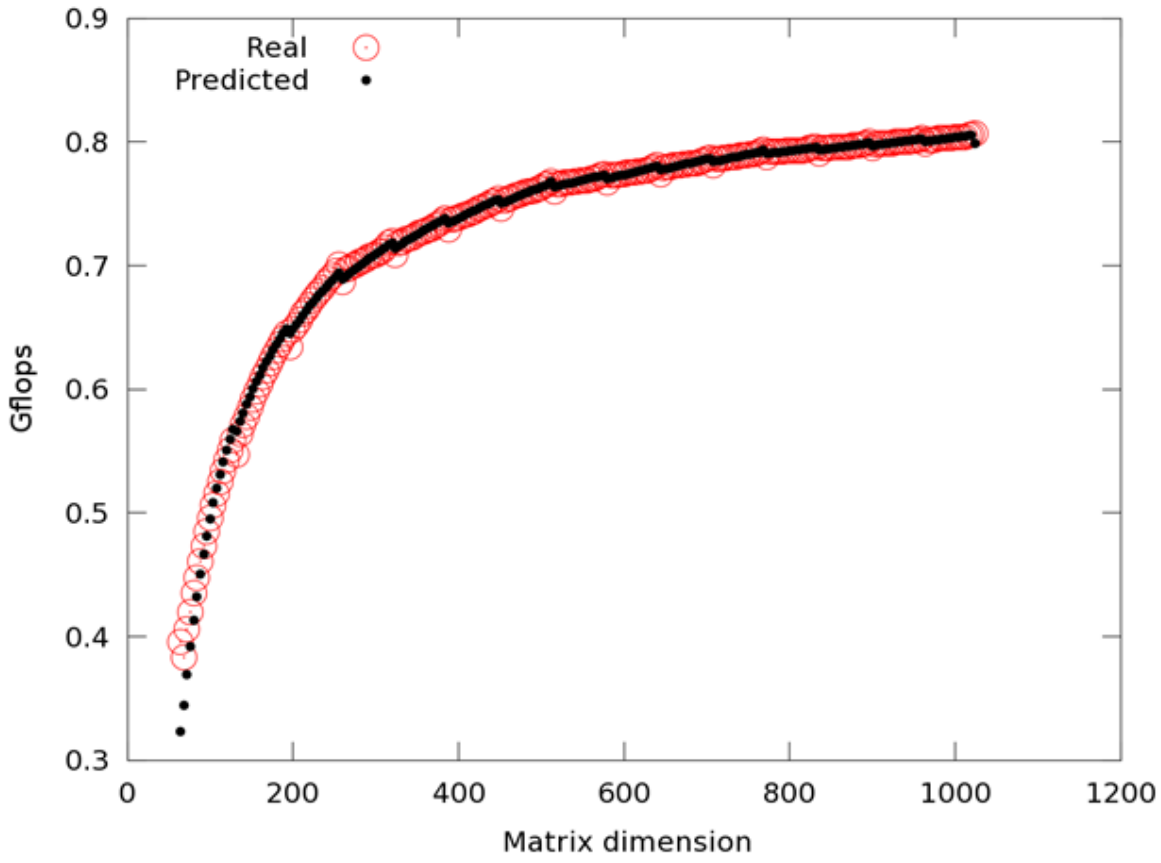


Figure 6: Real performance vs Predicted performance for one SHAVE

As it can be seen, the model can be adapted to other architectures by changing the necessary architecture parameters and, possibly, the kernel parameters, in the most likely case the micro-kernel code has to be modified for a new SHAVE instruction set.

9 References

BLIS documentation available at: https://github.com/flame/blis