Redbooks
ibm.com/redbooks

Matrix-Multiply Assist Best Practices Guide

Puneeth Bhat

José Moreira

Satish Kumar Sadasivam

Power Systems

IBM

Redpaper

IBM Redbooks

Matrix-Multiply Assist Best Practices Guide

April 2021

REDP-5612-00

Note: Before using this information and the product it supports, read the information in “Notices” on page v.

First Edition (April 2021)

This edition applies to the Matrix-Multiply Assist (MMA) architecture introduced in Power ISA Version 3.1.

© Copyright International Business Machines Corporation 2021. All rights reserved.
Note to U.S. Government Users Restricted Rights -- Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

© Copyright IBM Corp. 2021.

Notices

This information was developed for products and services offered in the US. This material might be available from IBM in other languages. However, you may be required to own a copy of the product or product version in that language in order to access it.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send license inquiries, in writing, to:
IBM Director of Licensing, IBM Corporation, North Castle Drive, MD-NC119, Armonk, NY 10504-1785, US

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some jurisdictions do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM websites are provided for convenience only and do not in any manner serve as an endorsement of those websites. The materials at those websites are not part of the materials for this IBM product and use of those websites is at your own risk.

IBM may use or distribute any of the information you provide in any way it believes appropriate without incurring any obligation to you.

The performance data and client examples cited are presented for illustrative purposes only. Actual performance results may vary depending on specific configurations and operating conditions.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

Statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to actual people or business enterprises is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. The sample programs are provided "AS IS", without warranty of any kind. IBM shall not be liable for any damages arising out of your use of the sample programs.

© Copyright IBM Corp. 2021.

Trademarks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corporation, registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the web at “Copyright and trademark information” at http://www.ibm.com/legal/copytrade.shtml

The following terms are trademarks or registered trademarks of International Business Machines Corporation, and might also be trademarks or registered trademarks in other countries.

Redbooks (logo)  ®	IBM Research®	POWER7®
IBM®	POWER®	Redbooks®

The following terms are trademarks of other companies:

Other company, product, or service names may be trademarks or service marks of others.

Preface

This publication is for software developers who want to understand the Matrix-Mulitply Assist (MMA) function, particularly those writing libraries for high performance computing and machine learning applications.

Authors

This paper was produced by the following team of specialists.

Puneeth Bhat A H is a Senior Performance Analyst working on Power Systems performance at IBM. Presently he is driving the cognitive workload and interpreter performance innovations for Power processors. Puneeth, with 10 years of IBM experience, has expertise in the areas of processor micro-architecture performance, compiler optimizations and performance instrumentation tools development. He holds a Bachelor degree from Visvesvaraya Technological University.

José E. Moreira is a Distinguished Research Staff Member in the Scalable Systems Department at the Thomas J. Watson Research Center. He received a B.S. degree in physics and B.S. and M.S. degrees in electrical engineering from the University of Sao Paulo, Brazil, in 1987, 1988, and 1990, respectively. He also received a Ph.D. degree in electrical engineering from the University of Illinois at Urbana-Champaign in 1995. Since joining IBM at the Thomas J. Watson Research Center, he has worked on a variety of high-performance computing projects. He was system software architect for the Blue Gene/L supercomputer and chief architect of the Commercial Scale Out project. He currently leads the IBM Research® work on the architecture of POWER® processor. He is an author or coauthor of over 100 technical papers and 15 US patents. Dr. Moreira is a Fellow of the IEEE (Institute of Electrical and Electronics Engineers) and a Distinguished Scientist of the ACM (Association for Computing Machinery).

Satish Kumar Sadasivam is a Senior Performance Architect who leads the workload characterization and future architecture design space exploration team in IBM. He currently focuses on exploring architectural and microarchitectural innovations for Enterprise AI and cloud centric workloads with primary focus towards optimizing the core for single thread, core throughput and compute performance. He has worked on IBM POWER processor architecture since POWER5+ timeline. He is one of the key contributors for the POWER10 processor key architecture features like Instruction Fusion, Branch prediction and Matrix Math Assist (MMA). He has more than 16 years of experience in the areas of workload characterization, microarchitecture design exploration, compiler code generation and optimization, competitive evaluation, post-silicon hardware bring-up and validation. He received an MS in computer science from the Madras Institute of Technology. He is an IBM master inventor. He has filed more than 25+ patents, published several papers and delivered several talks in his field of work.

© Copyright IBM Corp. 2021.

Now you can become a published author, too!

Here's an opportunity to spotlight your skills, grow your career, and become a published author—all at the same time! Join an IBM Redbooks® residency project and help write a book in your area of expertise, while honing your experience using leading-edge technologies. Your efforts will help to increase product acceptance and customer satisfaction, as you expand your network of technical contacts and relationships. Residencies run from two to six weeks in length, and you can participate either in person or as a remote resident working from your home base.

Find out more about the residency program, browse the residency index, and apply online at:

ibm.com/redbooks/residencies.html

Comments welcome

Your comments are important to us!

We want our papers to be as helpful as possible. Send us your comments about this paper or other IBM® Redbooks publications in one of the following ways:

- Use the online **Contact us** review Redbooks form found at:

  ibm.com/redbooks
- Send your comments in an email to:

  redbooks@us.ibm.com
- Mail your comments to:

  IBM Corporation, IBM Redbooks
  Dept. HYTD Mail Station P099
  2455 South Road
  Poughkeepsie, NY 12601-5400

Stay connected to IBM Redbooks

- Find us on Facebook:

  http://www.facebook.com/IBMRedbooks
- Follow us on Twitter:

  http://twitter.com/ibmredbooks
- Look for us on LinkedIn:

  http://www.linkedin.com/groups?home=&gid=2130806
- Explore new Redbooks publications, residencies, and workshops with the IBM Redbooks weekly newsletter:

  https://www.redbooks.ibm.com/Redbooks.nsf/subscribe?OpenForm
- Stay current on recent Redbooks publications with RSS Feeds:

  http://www.redbooks.ibm.com/rss.html

1

Matrix multiplication

Matrix multiplication is one of the most widely used compute kernels in a broad set of applications in the emerging fields of machine learning and deep learning. This document briefly describes Matrix-Multiply Assist (MMA), which is a newly developed architecture concept that was first introduced in Power ISA Version 3.1.

The fundamental architecture principles are explained with detailed instruction set usage, register file management concepts, and various supporting facilities. The goal of this document is to help the user grasp the concepts behind MMA. The use of MMA is shown in various code examples, both in basic code versions and in fully optimized code.

© Copyright IBM Corp. 2021.

1.1 Matrix-multiply operation

This section shows how a matrix multiplication is performed using a simple example. In this example, A and B are two 8x8 matrices, as shown in Figure 1-1. When you multiply matrices A and B, which are both 8x8 matrices, the resultant matrix C is also 8x8 in size.

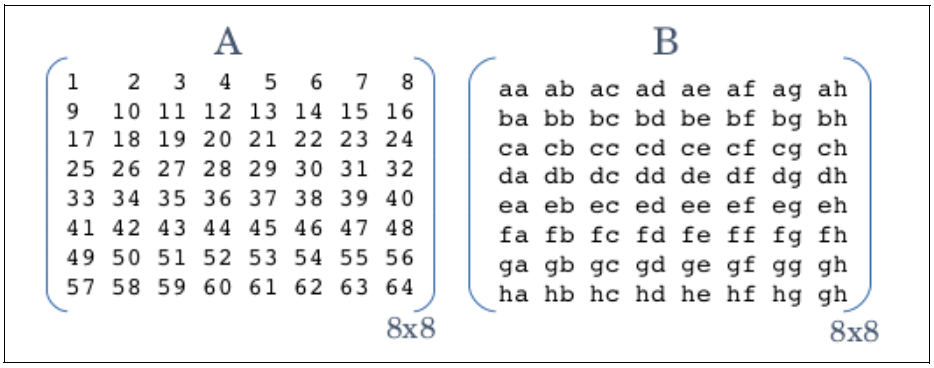

Figure 1-1 Matrices A and B used in multiplication

To generalize this concept for any size matrix, assume the size of matrix A is MxK (M rows and K columns) and size of matrix B is KxN (K rows and N columns). A basic requirement of the matrix multiplication operation is that the number of columns in matrix A should be the same as the number of rows in matrix B. If the number of rows of matrix A and number of columns of matrix B are different, then the matrix multiplication operation cannot be performed.

Figure 1-2 shows a simple example of how the matrix multiplication operation is performed. To generate one element of the output matrix C, which is of size 8x8, each element of the first row of matrix A is multiplied with corresponding element of the first column of matrix B. The results are accumulated, as shown in Figure 1-2.

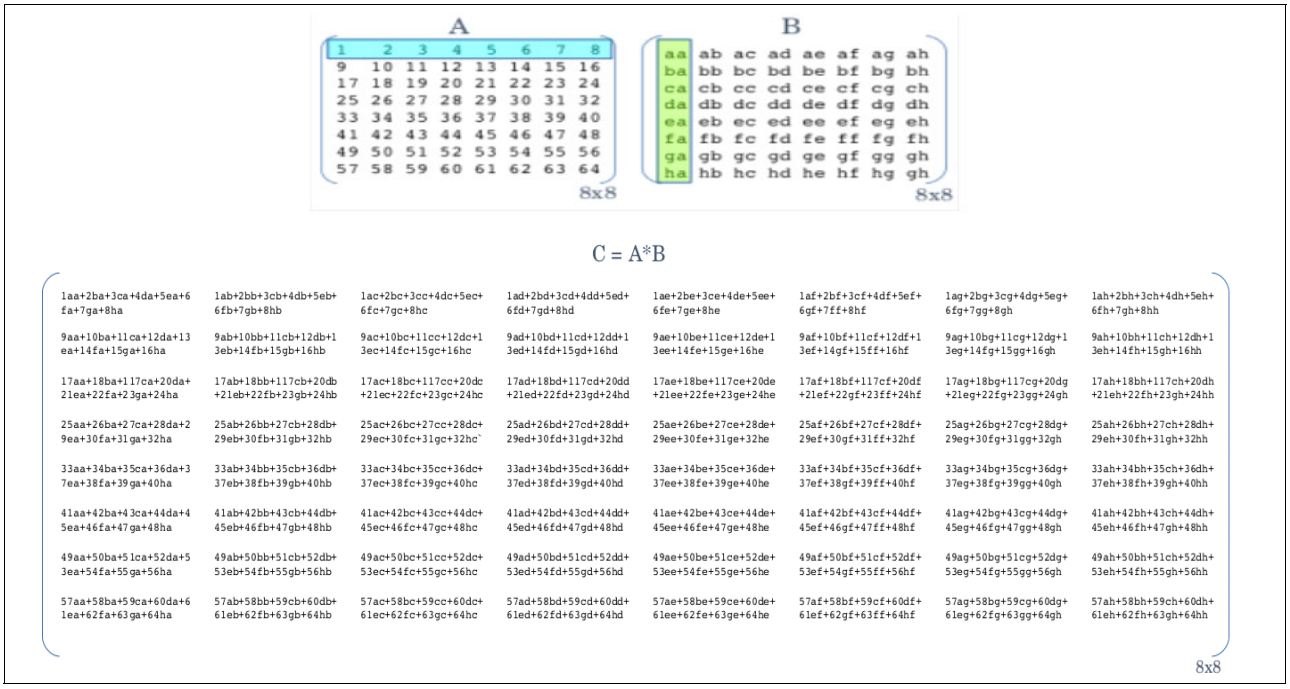

Figure 1-2 Example of an 8x8 matrix multiplication

The code used to perform this matrix multiplication is shown in Example 1-1. The code multiplies each element of i^{th} row of A with each element of j^{th} column of B. The result of each multiplication is accumulated to generate a result in the i^{th} row and j^{th} column of C.

Example 1-1 Basic matrix multiplication

```
#include <stdio.h>
#include <stdlib.h>

void printF (const char  *name, float *M, int m, int n) {
        printf ("\n**** Matrix %s****\n",name);
        for (int i=0; i< m; i++) {
                printf("|  ");
                for (int j=0; j< n; j++) printf("%-25.4f", *(M++));
                printf("   |\n");
        }
        printf("************************\n");
}

int main (int argc, char **argv ) {

    if (argc < 4) {
        printf("Usage: %s <M> <N> <K> \n", argv[0]);
        return -1;
    }

    const int M = atoi(argv[1]);
    const int N = atoi(argv[2]);
    const int K = atoi(argv[3]);

    printf("Running: %s M=%s N=%s K=%s \n", argv[0], argv[1], argv[2], argv[3]);

    float A[M][K];
    float B[K][N];
    float C[M][N];

    for (int i=0; i<M; i++) for (int j=0; j<N; j++) C[i][j] = 0;
    int x = 1;
    for (int i=0; i<M; i++) for (int j=0; j<K; j++) A[i][j] = float(x++) * 7 / 15;
    for (int i=0; i<K; i++) for (int j=0; j<N; j++) B[i][j] = float(x++) * 3 / 17;

    for (int i=0; i<M; i++) {
        for (int j=0; j<N; j++) {
            for (int k=0; k<K; k++)
                C[i][j] += A[i][k] * B[k][j];
        }
    }

    printF("C", (float *)C, M, N);
    return 0;
}
```

1.2 Vector outer product operation

In Figure 1-3, the code cannot be vectorized and it is not optimal. To optimize the matrix multiplication operation the vector outer product is performed. Matrix A is transposed and then the computation is performed in a blocked manner, as follows:

- The first 8x4 block of transposed matrix A (A^T) is iterated over the two 8x4 blocks of matrix B, computing outer products of the corresponding rows from blocks of A^T and B, to generate two 4x4 results.
- The second 8x4 block of matrix A transposed is iterated over the same two blocks of matrix B to generate the next two 4x4 results. This operation is explained in Figure 1-3.

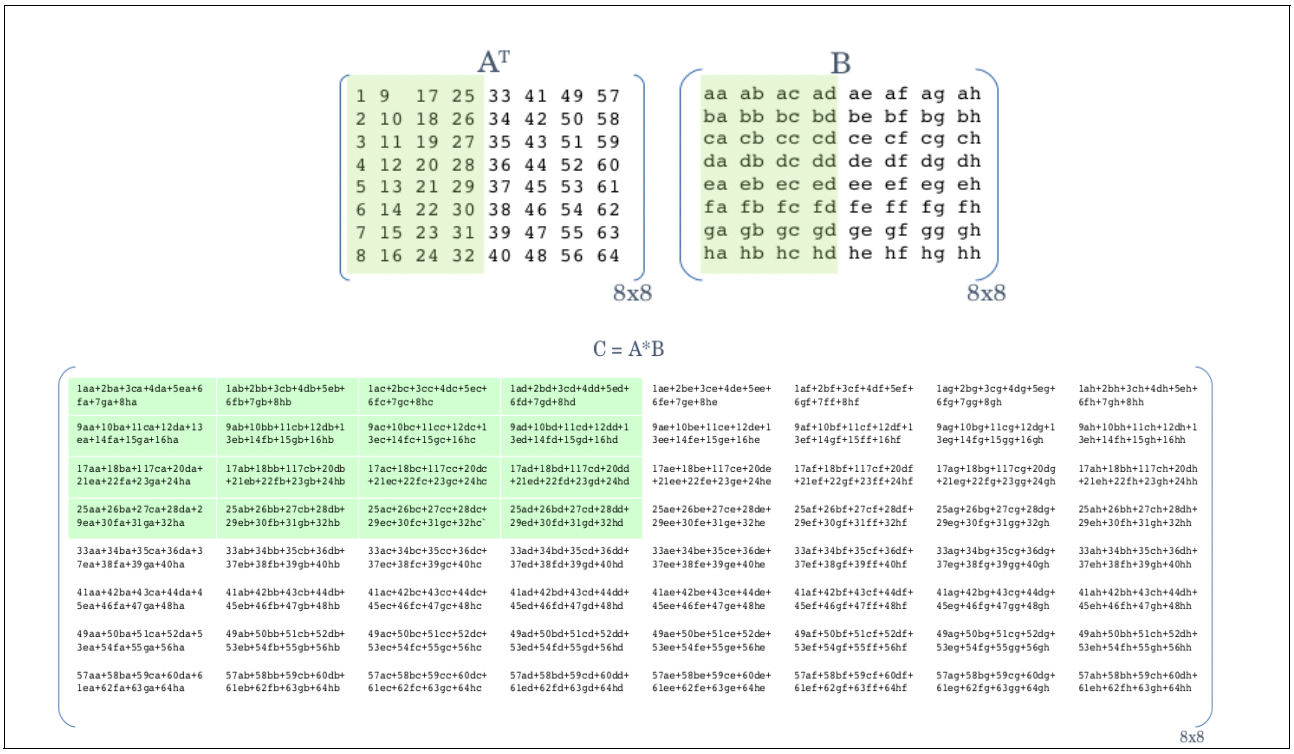

Figure 1-3 Matrix multiplication through outer products

Two key benefits of this optimization are:

- Opens up the possibility of parallelizing the entire computation.
- Reduces the number of loads performed and improves data locality because of reuse.

Figure 1-4 explains, in detail, how a partial matrix product is generated using the vector outer product operation. Each of the elements of transposed matrix A (1, 9, 17, 25) is multiplied with each of the elements of matrix B (aa, ab, ac, ad) to generate 16 outputs, which become the partial result of the resultant 4x4 output submatrix. The same operation is performed for each of the rows in both transposed matrix A and matrix B and the subsequent results are accumulated to get the final result.

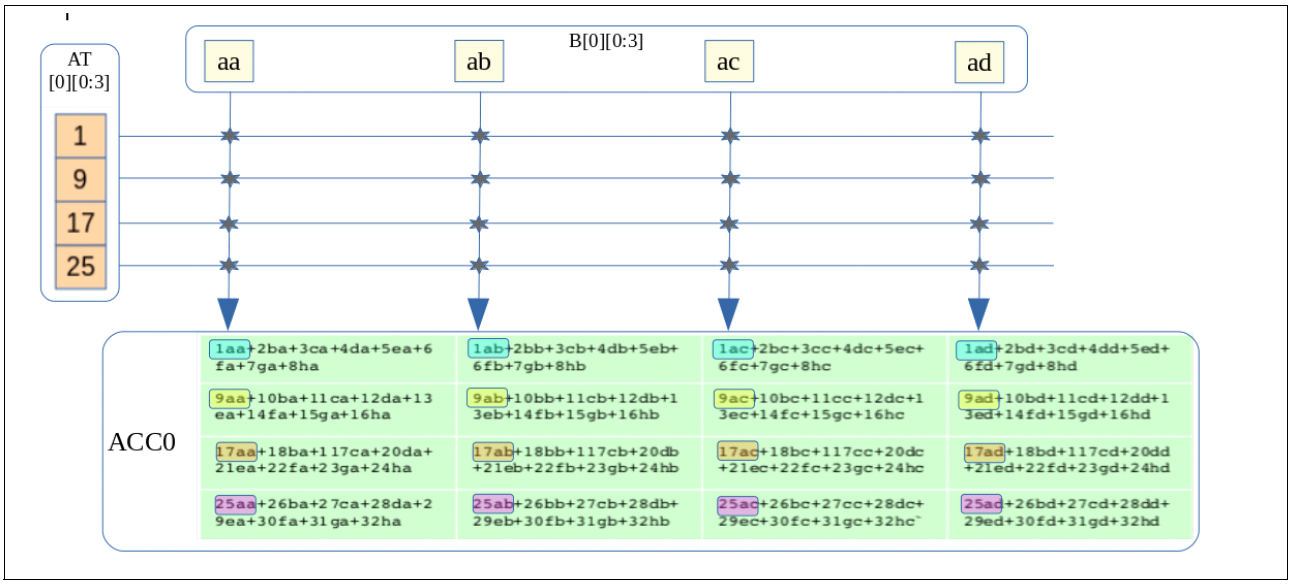

Figure 1-4 Generating a partial matrix product using a vector outer product

If this operation is repeated over all the blocks of both transposed matrix A and matrix B, then the final 8x8 results are generated, as shown in Figure 1-5. For a more detailed discussion on high-performance matrix multiplication, see Anatomy of High-Performance Matrix Multiply.

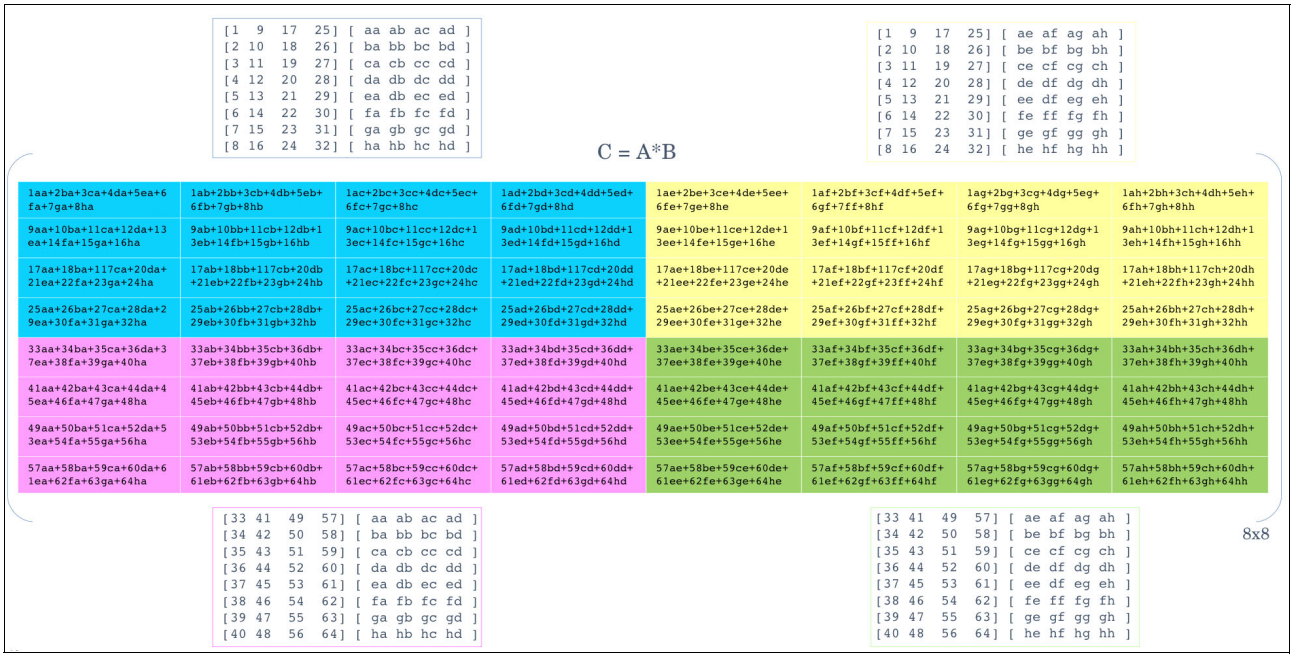

Figure 1-5 Full 8x8 matrix-multiply from four 4x4 blocks

1.3 Introduction to Vector Scalar Extension

Vector Scalar Extension (VSX) is the vector scalar extension capability introduced in the POWER ISA V2.06 and first implemented in the IBM POWER7® processor. In Figure 1-6, the VSX capability extends and unifies the 32 floating-point registers (FPRs) of Power ISA to 128 bits and combines with the existing 32 128-bit Vector Multimedia Extension (VMX) registers to create a single register file.

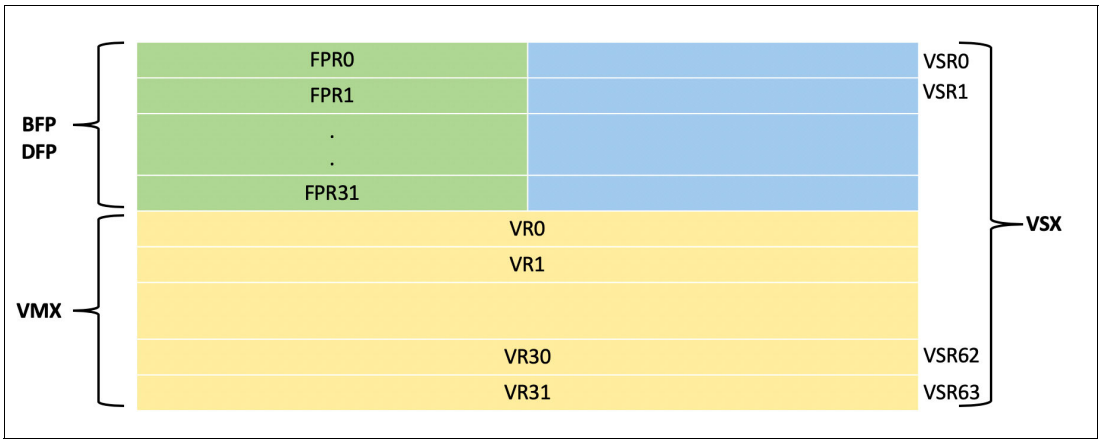

Figure 1-6 VSX registers

The VSX capability allows the VSX instruction to utilize 64 x 128-bit registers to perform its compute operations. VSX ISA supports numerous operations in both floating point and fixed point.

The key instructions that are useful in demonstrating the outer-product operations are:

- Vector loads and stores
- Splat instructions to replicate one element of the source vector register to all fields of the target vector register
- Vector floating point multiply-add instruction

1.4 Simple VSX code example for a vector outer product

Example 1-2 shows code that initializes matrices A, B, and C and gets a transform of matrix A.

Example 1-2 VSX Code example for a vector outer product

```
#include <stdio.h>
#include <stdlib.h>

#define KM 4
#define KN 4

extern "C" void sgemm_kernel_4x4(float*,float*,float*,int,int,int,int);

void sgemm(float *A, float *B, float *C, int M, int N, int K) {
    for (int i=0; i<M; i+=KM) {
    for (int j=0; j<N; j+=KN) {
        sgemm_kernel_4x4(A+i, B+j, C+j, K, M, N, N);
        }
    C += N*KM;
    }
}

void printF (const char  *name, float *M, int m, int n) {
        printf ("\n**** Matrix %s****\n",name);
        for (int i=0; i< m; i++) {
                printf("|  ");
                for (int j=0; j< n; j++) printf("%-25.4f", *(M++));
                printf("   |\n");
        }
        printf("************************\n");
}

int main (int    argc, char **argv ) {
    if (argc < 4) {
        printf("Usage: %s <M> <N> <K> \n", argv[0]);
        return -1;
   }

    const int M = atoi(argv[1]);
    const int N = atoi(argv[2]);
    const int K = atoi(argv[3]);

    printf("Running: %s M=%s N=%s K=%s \n", argv[0], argv[1], argv[2], argv[3]);

    float A[M][K];
    float AT[K][M];
    float B[K][N];
    float C[M][N];

    for (int i=0; i<M; i++) for (int j=0; j<N; j++) C[i][j] = 0;
    int x = 1;
    for (int i=0; i<M; i++) for (int j=0; j<K; j++) A[i][j] = float(x++) * 7 / 15;
    for (int i=0; i<K; i++) for (int j=0; j<N; j++) B[i][j] = float(x++) * 3 / 17;
    for (int i=0; i<M; i++) for (int j=0; j<K; j++) AT[j][i] = A[i][j];

    sgemm((float*)AT, (float*)B, (float*)C, M, N, K);

    printF("C", (float *)C, M, N);
    return 0;
}
```

The **Sgemm** routine computes the result matrix C in blocks of 4x4 (Power ISA VSX register size is 128 bits and contains 4x32-bit floating-point (fp32) elements.) The **sgemm_kernel_4x4** routine computes one 4x4 (KM x KN) block of the resultant matrix C.

The innermost loop of the kernel shown in Example 1-3 loads four elements from matrix A and four elements of matrix B and performs multiply-accumulate operations. This operation is performed '*K*' times to compute one full 4x4 block of the resultant matrix C.

Example 1-3 Multiply-accumulate operation

```
.section        ".text"
        .global sgemm_kernel_4x4
        .type   sgemm_kernel_4x4, @function

sgemm_kernel_4x4:
/* adjust lda, ldb, ldc for vector size 4 */
        slwi    7, 7, 2
        slwi    8, 8, 2
        slwi    9, 9, 2
 /* Reset VSX registers */
    xxlxor 0, 0, 0
    xxlxor 1, 1, 1
    xxlxor 2, 2, 2
    xxlxor 3, 3, 3
/* LOOP for K to 0 */
    K_LOOP:
 /* Load 4 elements of A, B */
       lxv    32, 0(3)
       lxv    33, 0(4)
 /* Copy each A[i] 4 times */
       xxspltw    34, 32, 3
       xxspltw    35, 32, 2
       xxspltw    36, 32, 1
       xxspltw    37, 32, 0
 /* Multiply-Add-Accumulate */
        xvmaddasp 0, 34, 33
        xvmaddasp 1, 35, 33
        xvmaddasp 2, 36, 33
        xvmaddasp 3, 37, 33
 /* Update Loop count & A,B */
        add     3, 3, 7
        add     4, 4, 8
        addic.  6, 6, -1
    bgt    K_LOOP
 /* Offsets of 4x4 C Matrix */
        slwi    3, 9, 1
        add     4, 5, 9
        add     6, 5, 3
        add     7, 4, 3
 /* Store the 4x4 c Matrix */
    stxv    0, 0(5)
    stxv    1, 0(4)
    stxv    2, 0(6)
    stxv    3, 0(7)
blr
```

The following command demonstrates how to build this example with the main C file linked with a defined assembly function using the latest MMA-supported GNU Compiler Collection (GCC):

```
>> g++ -mcpu=power10 -O2 sgemm_4x4.cc sgemm_vsx_kernel.s -o sgemm_vsx
```

2

Matrix-Multiply Assist Architecture

The Matrix-Multiply Assist (MMA) architecture is introduced in Power ISA v3.1. Several new concepts are described in this chapter, such as:

- An introduction of accumulator registers
- New compute instructions for the matrix multiplication operation
- Support for lower precision arithmetic beyond single- and double-precision

© Copyright IBM Corp. 2021.

2.1 Data types

MMA architecture supports both floating-point and integer data types. This support was introduced because of the growing requirements of future AI-inferencing models.

The following data types are floating-point:

- FP32 (IEEE single-precision)[1]
- FP64 (IEEE double-precision)[2]
- FP16 (IEEE half-precision)[3]
- bfloat16[4]

The following data types are integer:

- INT16 (16-bit integer)
- INT8 (8-bit integer)
- INT4 (4-bit integer)

2.2 Data layout in accumulators and VSRs

One of the key concepts of the MMA architecture is the accumulator register. There are eight such registers in Power ISA v3.1 and each accumulator is 512 bits. Currently, a clear association exists between accumulators and VSRs. Each VSR is 128 bits and four such VSRs are combined and shadowed to form one accumulator register. The first 32 VSRs are mapped to eight accumulator registers as shown in Figure 2-1 on page 12.

When programming with MMA instructions, one of the first design decisions is how to partition the space of VSRs and accumulators. It is important to keep the two separated. That is, if accumulator ACC*x* is being used (where *x* is from 0 - 7) from the accumulators. See 2.3, “Instructions” on page 13 for information on how to use instructions to transfer data.

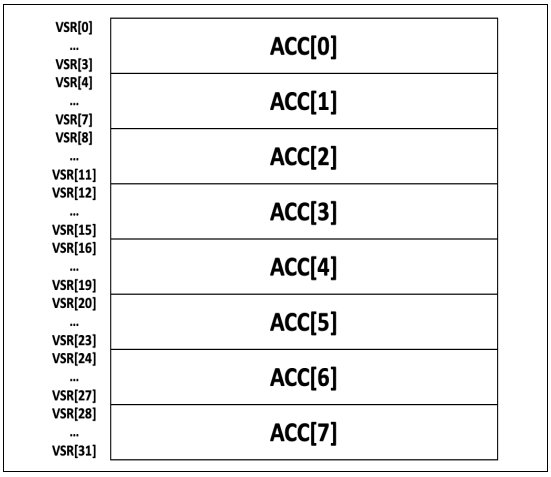

Figure 2-1 Accumulator architecture in POWER ISA v3.1

[1] 754-2019 - IEEE Standard for Floating-Point Arithmetic, found at: https://ieeexplore.ieee.org/document/8766229
[2] Ibid.
[3] Ibid.
[4] A transprecision floating-point platform for ultra-low power computing, found at: https://ieeexplore.ieee.org/abstract/document/8342167

Important: Do not mix instructions that use an accumulator (for example, ACC0) and the corresponding VSRs (for example, VSR[0:3]). You can mix instructions that use an accumulator (for example, ACC0) and VSRs that do not overlap it (for example, VSR[4:7]). VSR[32:63] should never overlap with an accumulator. This guideline is essential for both performance and guaranteed compatibility with future implementations.

2.3 Instructions

MMA architecture instructions are split into the following two categories:

- Accumulator operation instructions
- Outer product instructions

2.3.1 Accumulator operation instructions

The first category of instructions is those that deal with moving values between the VSRs and their associated accumulator registers. The following three instructions are used to operate on the accumulator registers:

- **`xxmfacc`**: Moves the contents from the accumulator to the associated VSR.
- **`xxmtacc`**: Moves the contents from the associated VSR to the accumulator.
- **`xxsetaccz`**: Zeros-out the contents of the accumulator.

When the move is done, both VSRs and accumulator registers are tied and the VSRs' content becomes undefined. If an instruction tries to write content to VSRs, the accumulator content becomes undefined. When the MMA operations are done, the **`xxmfacc`** instruction is used to copy the content of the accumulator back to the VSRs and the VSRs become valid. For more information, see Power ISA Version 3.1.

Examples are:

- **`xxmfacc AS`**

  AS can be any value from 0 - 7, each referring to one accumulator register. The **`xxmfacc AS`** instruction moves the contents of accumulator AS to the corresponding 4 VSRs.

- **`xxmtacc AT`**

  AT can be any value from 0 - 7, each referring to one accumulator register. The **`xxmtacc AT`** instruction moves the contents of 4 corresponding VSRs to accumulator AT.

- **`xxsetaccz AT`**

  AT can be any value from 0-7, each referring to one accumulator register. The **`xxsetaccz AT`** instruction sets the contents of accumulator **`AT`** to zero.

2.3.2 Outer product instructions

The second category of instructions is those that are used to perform the actual arithmetic. Both integer and floating point arithmetic are supported in the MMA architecture at different precision levels as described in 2.1, “Data types” on page 12.

Instructions for 32-bit floating-point arithmetic

Two 32-bit FP arithmetic instructions are used to discuss the functionality of MMA. The two instructions that are used to perform a single precision matrix multiplication operation are: **xvf32ger** and **xvf32gerpp**.

The difference between the **ger** instruction and the **gerpp** instruction is as follows:

- The **gerpp** instruction *accumulates* the results in the accumulator register. This instruction requires the accumulator to already have a defined content.
- The **ger** instruction *overwrites* the results in the accumulator register. This instruction defines the content of an accumulator, similar to the **xxmtacc** and **xxsetaccz** instructions.

xvf32gerpp AT,XA,XB, where:

- **AT** refers to any of the eight accumulator registers (ACC0-ACC7).
- **XA** and **XB** refer to VSRs.

For the **xvf32gerpp AT,XA,XB** instruction, assume **AT=1**, **XA= 32**, and **XB=33**. The VSR 32 has four 32-bit single precision values and VSR 33 has four 32-bit single precision values. Each value in VSR 32 is multiplied with each value in VSR 33, generating a 4×4 array of 32-bit results (a total of 512 bits of output). The output is accumulated with the content of ACC1, as shown in Figure 2-2.

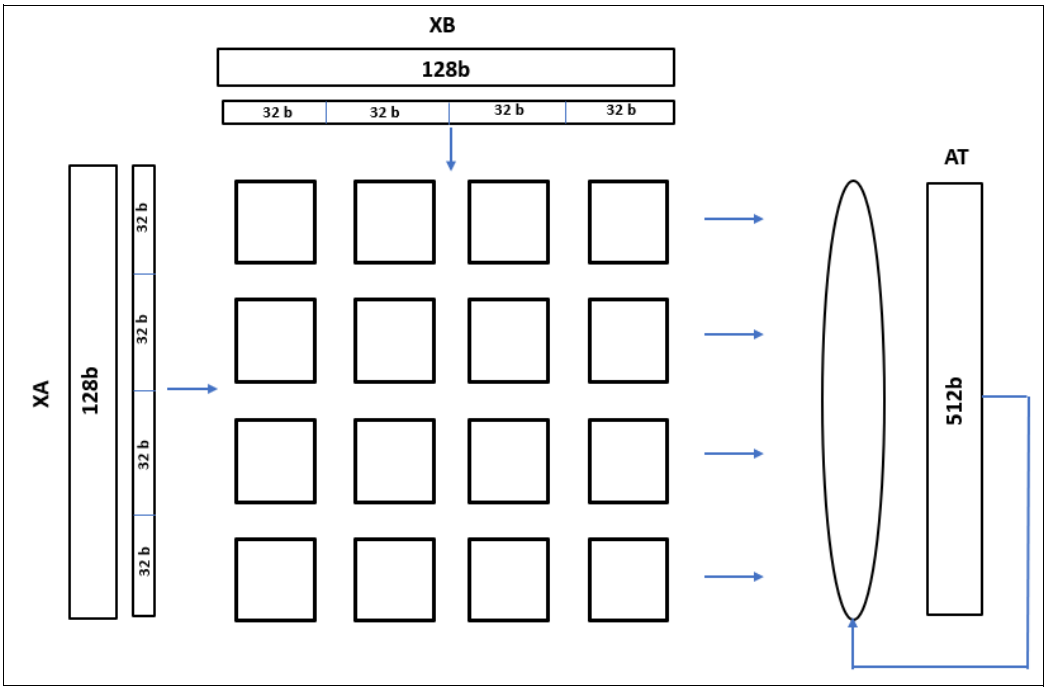

Figure 2-2 MMA xvf32gerpp instruction operation

Instructions for 8-bit arithmetic

MMA supports 8-bit integer operations. The 128-bit VSR register is split into 16 8-bit values. `vi8ger4` and `xvi8ger4pp` are the two instructions that perform outer product operation on the 8-bit values.

`xvi8ger4pp AT,XA,XB`, where:

- **`AT`** refers to any of the eight accumulator registers (ACC0-ACC7).
- **`XA`** and **`XB`** refer to VSR registers.

Assume that **`AT=1`**, **`XA=32`**, and **`XB=33`**. VSR 32 has sixteen 8-bit integer values and VSR 33 has sixteen 8-bit integer values. The function of the 8-bit outer product instruction is a bit different than the 32-bit arithmetic. Each four 8-bit value in a word of XA is multiplied with each corresponding four 8-bit value in a word of XB, and the four partial products are added together to produce a 32-bit result.

The output generated by the **`xvi8ger4pp`** instruction is shown in Figure 2-3.

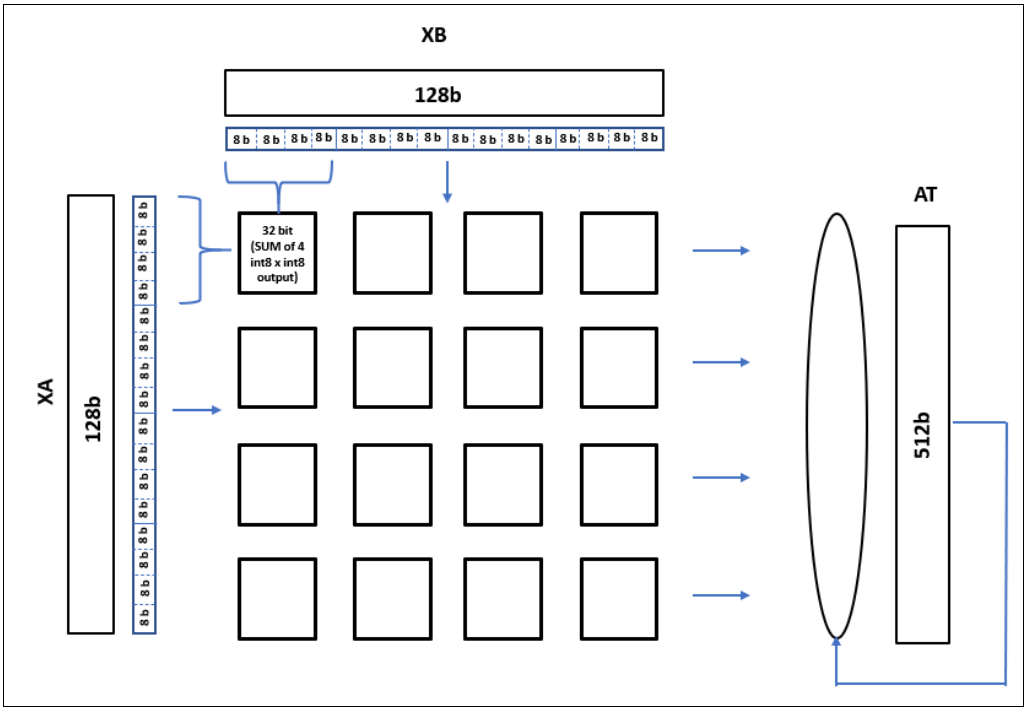

Figure 2-3 MMA xvi8ger4pp instruction operation

Though there are 16 8-bit values in each input VSR, the output generated still consists of 4x4 32-bit numbers and is 512 bits. The first four 8-bit elements are multiplied individually and the result of all four multiply operations is summed to generate one 32-bit value. Since there are 16 32-bit values produced, the result is still 512 bits. To use this instruction to accomplish an outer product operation, the input matrix needs to be reordered. The value formatting and how the outer product operation is performed is explained in detail in 3.3, “Mixed and lower precision matrix multiplication with MMA” on page 24.

2.3.3 Advanced feature: Lane masking

Lane masking is one of the advanced features available with the MMA architecture. The purpose of this feature is to perform an operation of a lower-sized input or to skip certain elements. For example, a 6x6 matrix multiplication as shown in Figure 2-4.

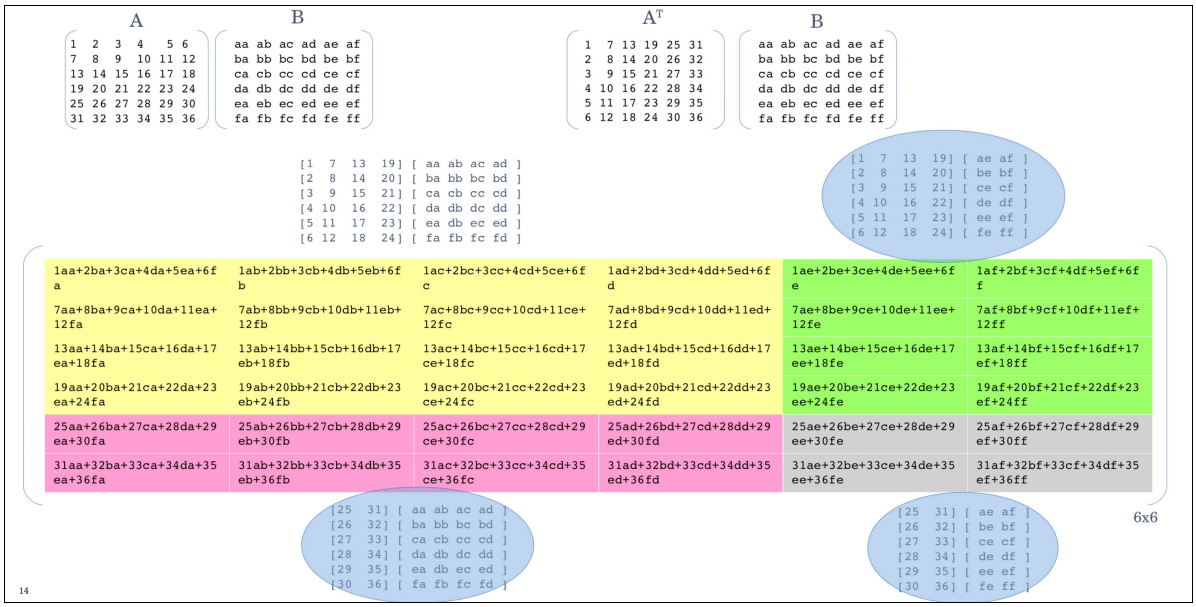

Figure 2-4 Example of a 6x6 matrix multiplication

Figure 2-5 on page 16 shows the details of the following single precision lane-masking instruction:

```
pmxvf32gerpp AT,XA,XB,XMSK,YMSK
```

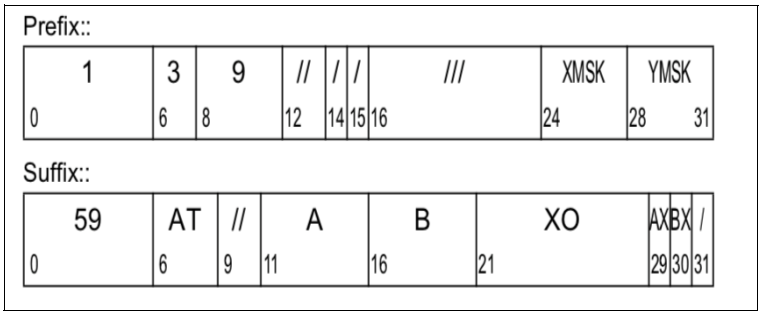

Figure 2-5 Instruction word details of a prefix instruction (pmxvf32gerpp)

The instruction encoding is 64 bits. The *prefix* is the first 32 bits and the *suffix* is the next 32 bits. The prefix architecture is a new capability introduced in the Power ISA v3.1 to extend the capability of the previous 32-bit fixed instruction-size architecture. This architecture is helpful when you represent bigger instructions with more parameters. Power ISA v3.1 uses the prefix architecture in several categories of instruction.

In Figure 2-5, the MMA lane-masking instruction has a total of five input arguments in the prefix:

- The first three arguments (`AT`, `XA`, and `XB`) are the same as the regular 32-bit single-precision MMA instruction.

- The last two arguments (`XMSK` and `YMSK`) are the mask values for the input VSR XA and XB.

The mask values for this instruction are of size 4 bits each. Each bit masks represents one 32-bit value in the source register. The multiplication operation is performed only if both mask bits of the respective input element are set to **1**. Otherwise, the respective results will be **0**.

For example, the output of the following command is shown in Figure 2-6:

```
pmxvf32gerpp 1,32,33,0xE,0xF
```

In Figure 2-6, the last 32-bit element of input register VSR[32] is skipped from the computation and the respective output to be accumulated in the ACC register 1 is `0`.

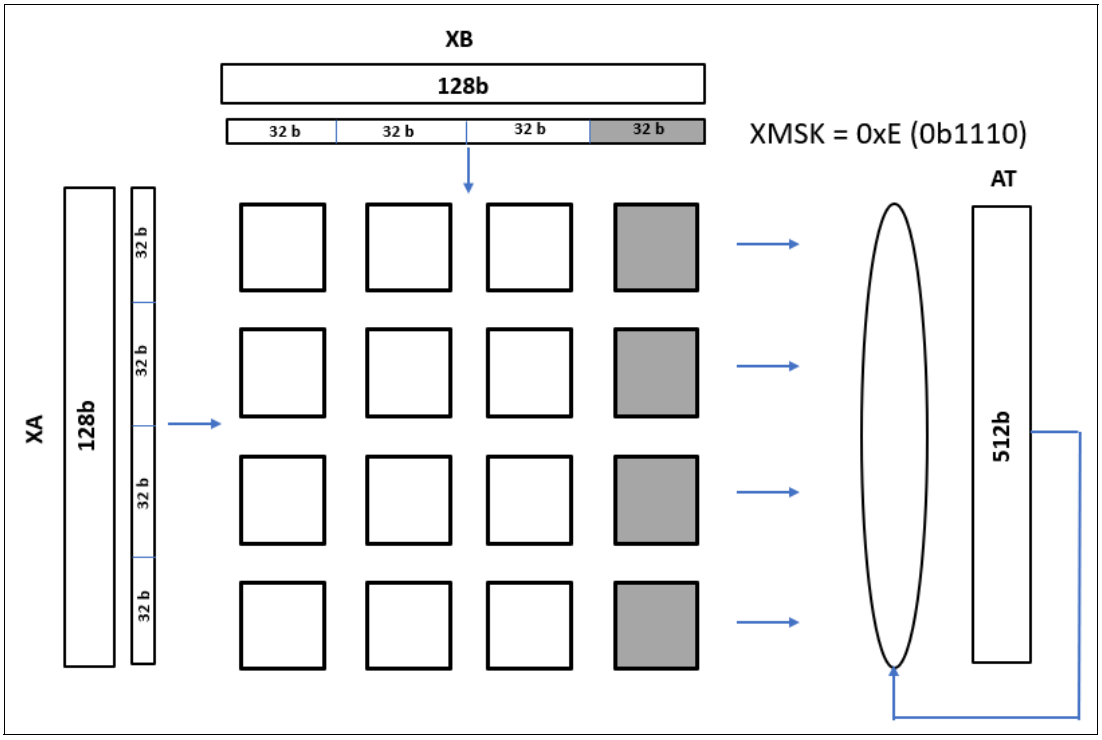

Figure 2-6 MMA pmxvf32gerpp instruction operation

The gray output shown in Figure 2-6 is not computed and the corresponding four 32-bit elements in the accumulator register remain unchanged.

3

Programming with Matrix-Multiply Assist

The Matrix-Multiply Assist (MMA) implementation of various kernels at different levels of precision is described in this chapter. The implementations that are shown use a single accumulator.

© Copyright IBM Corp. 2021.

3.1 Single-precision GEMM using MMA

The innermost kernel of **sgemm_kernel_4x4** shown in Example 3-1 loads four elements of A, loads four elements of B, and performs an outer product MMA operation to produce one 4x4 partial result of C in one accumulator register.

Example 3-1 SGEMM kernel using MMA instructions

```
.section        ".text"
        .global sgemm_kernel_4x4
        .type   sgemm_kernel_4x4, @function

sgemm_kernel_4x4:
/* adjust lda, ldb, ldc for vector size 4 */
        slwi    7, 7, 2
        slwi    8, 8, 2
        slwi    9, 9, 2
 /* Reset accumulator */
    xxsetaccz 0
/* LOOP for K to 0 */
    K_LOOP:
 /* Load 4 elements of A, B */
       lxv    32, 0(3)
       lxv    33, 0(4)
 /* Multiply-Add-Accumulate */
       xvf32gerpp 0, 32, 33
 /* Update Loop count & A,B */
        add     3, 3, 7
        add     4, 4, 8
        addic.  6, 6, -1
    bgt    K_LOOP
 /* Unprime the accumulator 0 */
    xxmfacc 0
 /* Offsets of 4x4 C Matrix */
        slwi    3, 9, 1
        add     4, 5, 9
        add     6, 5, 3
        add     7, 4, 3
 /* Store the 4x4 C Matrix */
    stxv    0, 0(5)
    stxv    1, 0(4)
    stxv    2, 0(6)
    stxv    3, 0(7)
blr
```

3.2 Double-precision GEMM using MMA and one accumulator

Each accumulator can hold eight (4×2) double-precision values and each VSR register (128-bit long) can contain two double-precision values (64-bit each). To produce an outer product with eight 64-bit values, we need three source VSRs so that four elements from A matrix can be multiplied with two elements from B matrix to generate a 4x2 result C submatrix. MMA enables this by taking a paired VSX register as a first operand and a single VSX register as a second operand.

For a double-precision **ger** instruction, the first operand is always an even VSX register and is considered as being paired with the next register.

Example 3-2 shows a simple example of a double-precision **gemm**. The code snippet shows the initialization of the input matrices and how an external `dgemm` kernel is referenced.

Example 3-2 dgemm example using 4x2 MMA kernel

```
#include <stdio.h>
#include <stdlib.h>

#define KM 4
#define KN 2

extern "C" void dgemm_kernel_4x2 (double *, double *, double *, int, int, int,
int);

void dgemm(double *A, double *B, double *C, int M, int N, int K) {
    for (int i=0; i<M; i+=KM) {
    for (int j=0; j<N; j+=KN) {
        dgemm_kernel_4x2(A+i, B+j, C+j, K, M, N, N);
        }
    C += N*KM;
    }
}

void printD (const char  *name, double *M, int m, int n) {
        printf ("\n**** Matrix %s****\n",name);
        for (int i=0; i< m; i++) {
                printf("|  ");
                for (int j=0; j< n; j++) printf("%-25.4f", *(M++));
                printf("   |\n");
        }
        printf("************************\n");
}

int main (int argc, char **argv ) {

    if (argc < 4) {
        printf("Usage: %s <M> <N> <K> \n", argv[0]);
        return -1;
    }

    const int M = atoi(argv[1]);
    const int N = atoi(argv[2]);
    const int K = atoi(argv[3]);
```

```
    printf("Running: %s M=%s N=%s K=%s \n", argv[0], argv[1], argv[2], argv[3]);

    double A[M][K];
    double AT[K][M];
    double B[K][N];
    double C[M][N];

    for (int i=0; i<M; i++) for (int j=0; j<N; j++) C[i][j] = 0;
    int x = 1;
    for (int i=0; i<M; i++) for (int j=0; j<K; j++) A[i][j] = double(x++) * 7 /
15;
    for (int i=0; i<K; i++) for (int j=0; j<N; j++) B[i][j] = double(x++) * 3 /
17;
    for (int i=0; i<M; i++) for (int j=0; j<K; j++) AT[j][i] = A[i][j];

    dgemm((double*)AT, (double*)B, (double*)C, M, N, K);

    printD("C", (double *)C, M, N);
    return=0;
```

The code in Example 3-3 defines the full routine of the **dgemm** 4x2 kernel in assembly using MMA instructions. This example can be compiled using the following command:

```
>> g++ -mcpu=power10 -O2 dgemm_kernel_4x2.s dgemm_4x2.cc -o dgemm_mma
```

Example 3-3 A simple 4x2 dgemm kernel using MMA instructions

```
.section        ".text"
        .global dgemm_kernel_4x2
        .type   dgemm_kernel_4x2, @function

dgemm_kernel_4x2:
/* adjust lda, ldb, ldc for vector size 8 */
        slwi    7, 7, 3
        slwi    8, 8, 3
        slwi    9, 9, 3

/* Reset acc0  */
    xxsetaccz    0

/* LOOP for K to 0 */
    K_LOOP:

/* Load 4 elements of A, B */
       lxvp    32, 0(3)
       lxv     34, 0(4)
/* Multiply-Add-Accumulate */
        xvf64gerpp 0, 32, 34

/* Update Loop count & A,B */
        add     3, 3, 7
        add     4, 4, 8
        addic.  6, 6, -1
    bgt    K_LOOP
/* Unprime the accumulator 0 */
    xxmfacc 0
```

```
/* Offsets of 4x2 C Matrix */
        slwi    3, 9, 1
        add     4, 5, 9
        add     6, 5, 3
        add     7, 4, 3
/* Store the 4x2 C Matrix */
    stxv    0, 0(5)
    stxv    1, 0(4)
    stxv    2, 0(6)
    stxv    3, 0(7)
blr
```

3.3 Mixed and lower precision matrix multiplication with MMA

With mixed precision matrix multiplication, the source matrices A and B are of reduced/low precision (either 8- or 16-bit, with the same type for A and B) and the resulting matrix C is of 32-bit integer elements. MMA supports these mixed precision matrix multiplications with a broad set of instructions, as explained in the ISA.

3.3.1 Source matrix reordering with Int8 as example

Consider an example of generating a 32-bit result matrix with two 8-bit lower-precision source matrices. Since VSX registers are 128 bits long, you can load 16 8-bit values from the two source matrices at each step. As the result matrix C type is 32-bit integer, accumulators still contain 16 elements (4x4 sub-matrix). The MMA operates slightly differently with lower-precision operations. Each set of four 8-bit values in each source VSX register is packed and considered as single unit. As a result, one source VSX register of 16 8-bit values is now assumed to hold four sets of four 8-bit values. Each set in the first VSX register is multiplied and accumulated with each set in second VSX register.

Within a set, the four 8-bit values are multiplied and accumulated in a one-to-one mapping, as described in Figure 3-1 on page 25 and Example 3-4 on page 25.

Figure 3-1 A single xvi8ger instruction performs a 4x4 matrix-multiply

Example 3-4 shows how the four 8-bit values are multiplied and accumulated in a one-to-one mapping.

Example 3-4 Code showing matrix reorganization for an int-8 gemm with MMA kernel

```
#include <stdio.h>
#include <stdlib.h>
#include <inttypes.h>

#define KM 4
#define KN 4
```

```
#define Q 4

extern "C" void i8gemm_mma_4x4 (int8_t *, uint8_t *, int32_t *, int, int, int, int);

void i8gemm(int8_t *A, uint8_t *B, int32_t *C, int M, int N, int K) {

    int8_t *At = (int8_t *) malloc(M*K);  //transform A[M][K] --> At[K/Q][M*Q]
    uint8_t *Bt = (uint8_t *) malloc(N*K); //transform B[K][N] --> At[K/Q][N*Q]

    for (int i=0,x=0; i<K; i+=Q) {
        for(int j=0; j<M; j++) {
             for(int l=0; l<Q; l++) At[x++] = *(A+(j*K)+l);
        }
        A = A+Q;
    }

    for (int i=0, x=0; i<K; i+=Q) {
        for(int j=0; j<N; j++) {
            for(int l=0; l<Q; l++) Bt[x++] = *(B+(l*N)+j);
        }
        B+=Q*N;
    }

    for (int i=0; i<M; i+=KM) {
    for (int j=0; j<N; j+=KN) {
        i8gemm_mma_4x4(At+(i*Q), Bt+(j*Q), C+j, K/Q, M, N, N);
        }
    C += N*KM;
    }
}

void printI (const char  *name, int32_t *M, int m, int n) {
        printf ("\n**** Matrix %s****\n",name);
        for (int i=0; i< m; i++) {
                printf("|  ");
                for (int j=0; j< n; j++) printf("%-25d", *(M++));
                printf("   |\n");
        }
        printf("************************\n");
}

int main (int    argc, char **argv ) {

    if (argc < 4) {
        printf("Usage: %s <M> <N> <K> \n", argv[0]);
        return -1;
    }

    const int M = atoi(argv[1]);
    const int N = atoi(argv[2]);
    const int K = atoi(argv[3]);

    printf("Running: %s M=%s N=%s K=%s \n", argv[0], argv[1], argv[2], argv[3]);

    int8_t A[M][K];
    uint8_t B[K][N];
    int32_t C[M][N];

    for (int i=0; i<M; i++) for (int j=0; j<N; j++) C[i][j] = 0;
    int x = 1;
```

```
    for (int i=0; i<M; i++) for (int j=0; j<K; j++) A[i][j] = (x++)%128;
    for (int i=0; i<K; i++) for (int j=0; j<N; j++) B[i][j] = (x++)%256;

    i8gemm((int8_t *)A, (uint8_t *)B, (int32_t *)C, M, N, K);

    printI("C", (int32_t *)C, M, N);
    return 0;
```

The code in Example 3-5 shows a simple 4x4 int-8 gemm kernel using MMA instructions.

Example 3-5 A 4x4 int-8 gemm kernel with xvi8ger MMA instruction

```
.section        ".text"
        .global i8gemm_mma_4x4
        .type   i8gemm_mma_4x4, @function

i8gemm_mma_4x4:
/* adjust lda, ldb, ldc for vector size 4 */
        slwi    7, 7, 2
        slwi    8, 8, 2
        slwi    9, 9, 2
 /* Reset acc0 & prime */
    xxsetaccz 0
/* LOOP for K to 0 */
    K_LOOP:
 /* Load 4 elements of A, B */
       lxv    32, 0(3)
       lxv    33, 0(4)
 /* Multiply-Add-Accumulate */
       xvi8ger4pp 0, 32, 33
 /* Update Loop count & A,B */
       add     3, 3, 7
       add     4, 4, 8
       addic.  6, 6, -1
    bgt    K_LOOP
 /* Unprime the accumulator 0 */
    xxmfacc 0
 /* Offsets of 4x4 C Matrix */
        slwi    3, 9, 1
        add     4, 5, 9
        add     6, 5, 3
        add     7, 4, 3
 /* Store the 4x4 C Matrix */
    stxv    0, 0(5)
    stxv    1, 0(4)
    stxv    2, 0(6)
    stxv    3, 0(7)
blr
```

4

Advanced programming concepts

This chapter describes how to optimize the matrix multiplication examples that are shown in this publication. It also explains some of the improved techniques for effectively using the compute units with load reuse and cache blocking.

© Copyright IBM Corp. 2021.

4.1 Multiple accumulators SGEMM for load value reuse

Consider the example of the **sgemm** instruction, shown in Example 3-4 on page 25, which uses one accumulator to generate a 4x4 result, where two load operations are required for each **gerpp** instruction. This type of model restricts the MMA unit utilization, as it is limited by the number of load ports. This limitation can be overcome by using multiple accumulators to reuse the same loaded data multiple times, as shown in Figure 4-1 and Example 4-1. When you use all eight accumulators and reuse loads, the result is eight **gerpp** instructions for every six vector register loads. (The code uses the register pair **load lxvp** instruction, which is another feature of Power ISA v3.1.) This kernel generates an 8x16 submatrix of C (eight 4x4 accumulators).

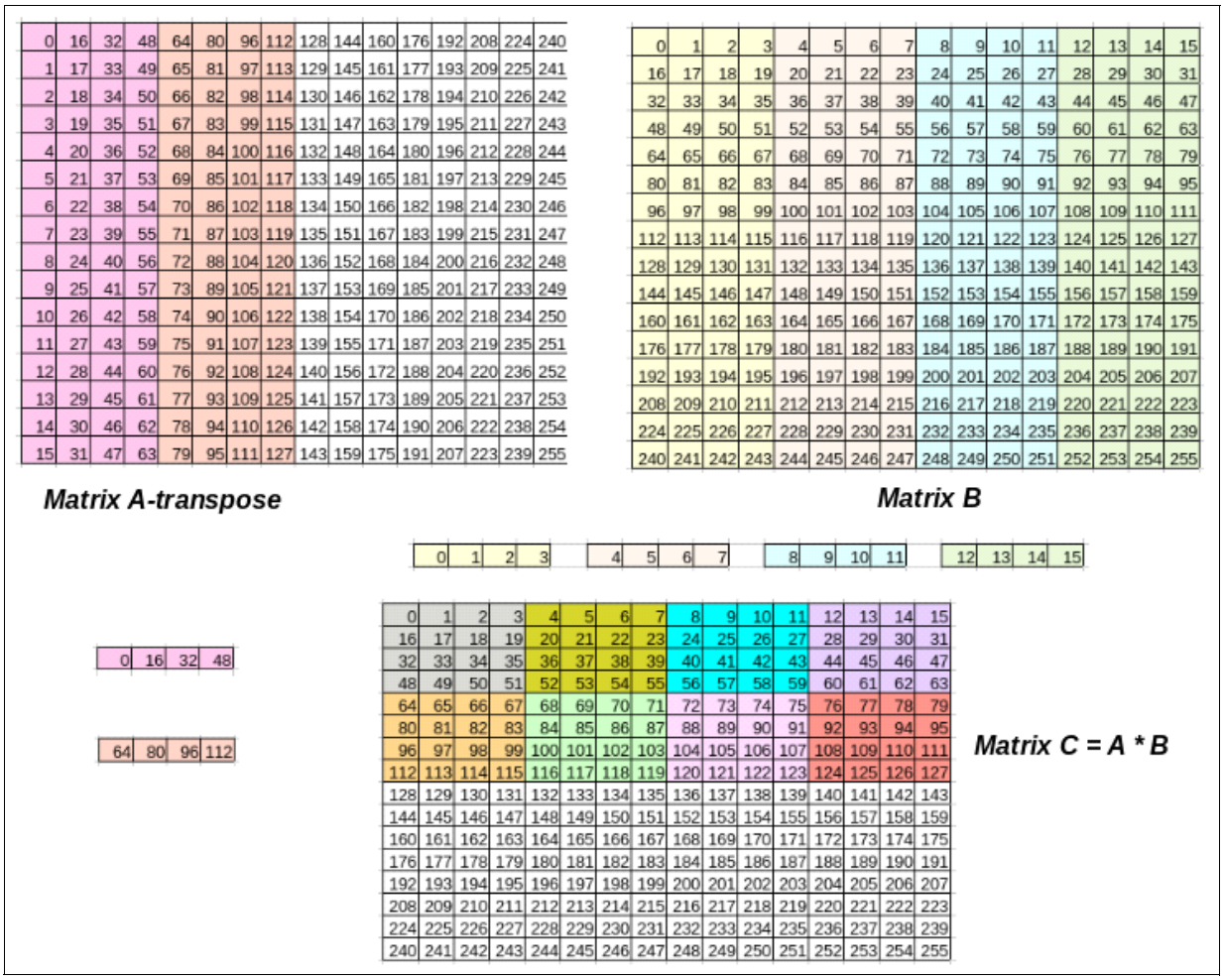

Figure 4-1 Example of an 8x16 GEMM FP32 kernel execution

Example 4-1 GEMM FP32 kernel execution code

```
#define KM 8
#define KN 16

extern "C" void sgemm_mma_8x16 (float*, float*, float*, int, int, int, int);

void sgemm(float *A, float *B, float *C, int M, int N, int K) {
    for (int i=0; i<M; i+=KM) {
    for (int j=0; j<N; j+=KN) {
        sgemm_mma_8x16(A+i, B+j, C+j, K, M, N, N);
        }
    C += N*KM;
    }
}
```

Example 4-2 shows a code snippet for a **sgemm** kernel that reuses the loads and all eight accumulators for improved compute unit utilization. The code computes a resultant 8x16 submatrix.

Example 4-2 An 8-accumulator version of sgemm kernel

```
.section        ".text"
        .global sgemm_mma_8x16
        .type   sgemm_mma_8x16, @function

sgemm_mma_8x16:
/* adjust lda, ldb, ldc for vector size 4 */
        slwi    7, 7, 2
        slwi    8, 8, 2
        slwi    9, 9, 2
 /* Reset acc0-7 & prime */
    xxsetaccz    0
    xxsetaccz    1
    xxsetaccz    2
    xxsetaccz    3
    xxsetaccz    4
    xxsetaccz    5
    xxsetaccz    6
    xxsetaccz    7
/* LOOP for K to 0 */
    K_LOOP:
 /* Load 4 elements of A, B */
       lxvp   32, 0(3)
       lxvp   34, 0(4)
       lxvp   36, 32(4)
 /* Multiply-Add-Accumulate */
       xvf32gerpp 0, 33, 35
       xvf32gerpp 1, 33, 34
       xvf32gerpp 2, 33, 37
       xvf32gerpp 3, 33, 36
       xvf32gerpp 4, 32, 35
       xvf32gerpp 5, 32, 34
       xvf32gerpp 6, 32, 37
       xvf32gerpp 7, 32, 36
 /* Update Loop count & A,B */
       add     3, 3, 7
       add     4, 4, 8
       addic.  6, 6, -1
```

```
    bgt    K_LOOP
/* Unprime the acc0-7 */
    xxmfacc 0
    xxmfacc 1
    xxmfacc 2
    xxmfacc 3
    xxmfacc 4
    xxmfacc 5
    xxmfacc 6
    xxmfacc 7
 /* Offsets of 4x4 C Matrix */
    slwi  3, 9, 1
    add   4, 5, 9
    add   6, 5, 3
    add   7, 4, 3
 /* Store the 4x16 c Matrix */
    stxv    3,  0(5)
    stxv    2,  0(4)
    stxv    1,  0(6)
    stxv    0,  0(7)
    stxv    7,  16(5)
    stxv    6,  16(4)
    stxv    5,  16(6)
    stxv    4,  16(7)
    stxv    11, 32(5)
    stxv    10, 32(4)
    stxv    9,  32(6)
    stxv    8,  32(7)
    stxv    15, 48(5)
    stxv    14, 48(4)
    stxv    13, 48(6)
    stxv    12, 48(7)
 /* Update index of C */
    add   5, 7, 9
    add   4, 5, 9
    add   6, 5, 3
    add   7, 4, 3
 /* Store the 4x16 c Matrix */
    stxv    19, 0(5)
    stxv    18, 0(4)
    stxv    17, 0(6)
    stxv    16, 0(7)
    stxv    23, 16(5)
    stxv    22, 16(4)
    stxv    21, 16(6)
    stxv    20, 16(7)
    stxv    27, 32(5)
    stxv    26, 32(4)
    stxv    25, 32(6)
    stxv    24, 32(7)
    stxv    31, 48(5)
    stxv    30, 48(4)
    stxv    29, 48(6)
    stxv    28, 48(7)
blr
```

4.2 Multiple accumulators DGEMM for load value reuse

From the DGEMM example and as specified in MMA design, a single 512-bit accumulator result of a double-precision instruction can hold a resultant 4x2 submatrix. To perform one double-precision MMA instruction, two VSR registers (two loads, four elements) from matrix A and one VSR register (one load, two elements) from matrix B. By using all eight accumulators, you can generate a resultant 8x8 result. Including the reusing of loads, a total of eight VSR loads (four each from matrices A and B) need to be performed. A simple example of an **dgemm_kernel_8x8** is shown in Example 4-3.

Example 4-3 8x8 FP64 GEMM kernel code

```
#define DKM 8
#define DKN 8

extern "C" void dgemm_kernel_8x8 (double *, double *, double *, int, int, int,
int);

void dgemm(double *A, double *B, double *C, int M, int N, int K) {
    for (int i=0; i<M; i+=DKM) {
    for (int j=0; j<N; j+=DKN) {
        dgemm_kernel_8x8(A+i, B+j, C+j, K, M, N, N);
        }
    C += N*DKM;
    }
}
```

The code snippet in Example 4-4 shows an improved **dgemm** kernel using eight accumulators. This code computes an 8x8 block of result matrix.

Example 4-4 An 8x8 dgemm kernel using 8 accumulators

```
         .section        ".text"
        .global dgemm_kernel_8x8
        .type   dgemm_kernel_8x8, @function

dgemm_kernel_8x8:
/* adjust lda, ldb, ldc for vector size 8 */
        slwi    7, 7, 3
        slwi    8, 8, 3
        slwi    9, 9, 3
 /* Reset acc0-7 & prime */
    xxsetaccz    0
    xxsetaccz    1
    xxsetaccz    2
    xxsetaccz    3
    xxsetaccz    4
    xxsetaccz    5
    xxsetaccz    6
    xxsetaccz    7
/* LOOP for K to 0 */
    K_LOOP:
 /* Load 4 elements of A, B */
       lxvp   32, 0(3)
       lxvp   34, 32(3)
```

```
        lxvp   36, 0(4)
        lxvp   38, 32(4)
   /* Multiply-Add-Accumulate */
        xvf64gerpp 0, 32, 37
        xvf64gerpp 1, 32, 36
        xvf64gerpp 2, 32, 39
        xvf64gerpp 3, 32, 38
        xvf64gerpp 4, 34, 37
        xvf64gerpp 5, 34, 36
        xvf64gerpp 6, 34, 39
        xvf64gerpp 7, 34, 38
   /* Update Loop count & A,B */
        add     3, 3, 7
        add     4, 4, 8
        addic.  6, 6, -1
     bgt    K_LOOP
  /* Unprime the acc0-7 */
      xxmfacc 0
      xxmfacc 1
      xxmfacc 2
      xxmfacc 3
      xxmfacc 4
      xxmfacc 5
      xxmfacc 6
      xxmfacc 7
   /* Offsets of 4x4 C Matrix */
      slwi  3, 9, 1
      add   4, 5, 9
      add   6, 5, 3
      add   7, 4, 3
   /* Store the 4x16 c Matrix */
      stxv    3,  0(5)
      stxv    2,  0(4)
      stxv    1,  0(6)
      stxv    0,  0(7)
      stxv    7,  16(5)
      stxv    6,  16(4)
      stxv    5,  16(6)
      stxv    4,  16(7)
      stxv    11, 32(5)
      stxv    10, 32(4)
      stxv    9,  32(6)
      stxv    8,  32(7)
      stxv    15, 48(5)
      stxv    14, 48(4)
      stxv    13, 48(6)
      stxv    12, 48(7)
   /* Update index of C */
      add   5, 7, 9
      add   4, 5, 9
      add   6, 5, 3
      add   7, 4, 3
   /* Store the 4x16 c Matrix */
      stxv    19, 0(5)
      stxv    18, 0(4)
```

```
    stxv    17, 0(6)
    stxv    16, 0(7)
    stxv    23, 16(5)
    stxv    22, 16(4)
    stxv    21, 16(6)
    stxv    20, 16(7)
    stxv    27, 32(5)
    stxv    26, 32(4)
    stxv    25, 32(6)
    stxv    24, 32(7)
    stxv    31, 48(5)
    stxv    30, 48(4)
    stxv    29, 48(6)
    stxv    28, 48(7)
         blr
```

Programming note: Instead of the resultant 8x8 submatrix shown in Example 4-4 on page 33, you can use an alternate approach to generate a resultant 4x16 submatrix for `dgemm` using eight accumulators. In this approach, the kernel requires ten vector loads, instead of eight vector loads. Though the kernel stresses the load units, there might be a slight advantage over the cache blocking method, which is discussed in 4.3, “SGEMM performance with advanced cache-blocking” on page 35.

4.3 SGEMM performance with advanced cache-blocking

Consider the multiplication of two float (single-precision) 256 x 256 matrices using an **`sgemm_kernel_8x16`**, which computes the resultant C matrix in 8x16 blocks. In this example, M=256, K=256, and N=256. To compute the full resultant 256 x 256 matrix C, the program loops over A in blocks of size 8x256 and over B in blocks of size 16x256, as shown in Example 4-5.

Example 4-5 Full computation of matrix C in blocks of size 8x16

```
#define KM 8
#define KN 16

void sgemm(float *A, float *B, float *C, int M, int N, int K) {
    for (int i=0; i<M; i+=KM) {
        for (int j=0; j<N; j+=KN) {
            sgemm_kernel_8x16(A+i, B+j, C+j, K, M, N, N);
        }
        C += N*KM;
    }
}
```

To carry out the computation of one 8x16 block of matrix C in a processing core (call to **`sgemm_kernel_8x16`** in the innermost kernel of Example 4-5), you need one 8x256 block of matrix A (8x256x4 bytes = 8 KiB) and one 16x256 block of matrix B (16x256x4 bytes = 16 KiB) to be loaded into the L1 cache. You can then compute one 8x16 block of resultant matrix C. In the second iteration of the innermost loop, the 8-KiB block of matrix A is retained and the next block of 16-KiB data of matrix B is freshly loaded onto L1 to compute the next 8x16 block of matrix C.

To improve the performance of the computation, retain the bigger block of data (from matrix B) in L1 cache and move through the smaller blocks (from matrix A). This is achieved by simply interchanging the loops, as shown in Example 4-6.

Example 4-6 Loops interchanged to keep the bigger block in L1 cache

```
#define KM 8
#define KN 16

void sgemm(float *A, float *B, float *C, int M, int N, int K) {
    for (int j=0; j<N; j+=KN) {
        for (int i=0; i<M; i+=KM) {
            sgemm_kernel_8x16(A+i, B+j, C+(N*KM), K, M, N, N);
        }
        C += KN;
    }
}
```

Assume an architecture with an L1 cache of 32 KiB. In the 256x256 matrix-multiplication example shown in Example 4-6, 16 KiB of matrix B and 8 KiB of matrix A are consumed by each execution of **`sgemm_kernel_8x16`**. The total of 24 KiB fits well within the L1 cache. Now consider an example of K=1024. The size of matrix B block becomes 16x1024x4 bytes = 64 KiB and the size of matrix A block grows to 32 KiB. The total exceeds the L1 cache size by a factor of three. Therefore, to keep the computation well within the L1 cache, there needs to be a third loop to iterate over 'K' in chunks of size KS. If KS=256 in this case, you need to repeat the above set of iterations K/KS = 1024/256 = 4 times.

5

Matrix-Multiply Assist programming with compiler built-ins

Support for Matrix-Multiply Assist (MMA) instructions and built-ins is enabled in the latest version of GCC and LLVM compilers, which are publicly available. You can use GCC 10 and later and LLVM 12.0 to build programs with MMA. Though the compiler does not yet generate MMA instructions from the direct existing source code, support is available to recognize inline MMA assembly instructions. The compilers can also generate binaries for **gemm** programs with MMA compiler built-ins.

For MMA programming, in addition to the already supported vector data types, the new `__vector_quad` data type is introduced to represent an accumulator. This data type represents a set of four vector registers forming an accumulator.

The following set of built-ins is used to move values to and from vector registers to accumulators:

- `void __builtin_mma_xxmtacc (__vector_quad *);`
- `void __builtin_mma_xxmfacc (__vector_quad *);`

The following built-in can be used to set the accumulator to zero:

```
void __builtin_mma_xxsetaccz (__vector_quad *);
```

The following built-ins can be used to collate and dismantle the accumulator register (`__vector_quad`) to four independent registers to be further used in the program:

- `void __builtin_mma_assemble_acc (__vector_quad *, vec_t, vec_t, vec_t, vec_t);`
- `void __builtin_mma_disassemble_acc (void *, __vector_quad *);`

The `builtin_mma_xv*` can be used to perform the matrix-multiply-and-accumulate operations. The following code is an example built-in for 32-bit floating-point `ger` operation:

```
void __builtin_mma_xvf32gerpp (__vector_quad *, vec_t, vec_t);
```

© Copyright IBM Corp. 2021.

5.1 Simple MMA SGEMM example using built-ins

Use the **gcc** compiler with the following command to compile the code example shown in Example 5-1. For a list of MMA compiler built-ins, see Appendix A, “List of Matrix-Multiply Assist compiler built-ins” on page 41.

```
> gcc -o sgemm -O2 -mcpu=power10 -mtune=power10 sgemm_intrinsics.cc
```

Example 5-1 Sample GEMM code using MMA compiler built-ins

```
#define KM 4
#define KN 4

typedef vector unsigned char    vec_t;
typedef __vector_quad   acc_t;

void sgemm_kernel_4x4 (float *a, float *b, float *c, int K, int lda, int ldb, int
ldc) {

        int i;
        vec_t vec_A, vec_B, vec_C[4];
        acc_t acc_0;

        __builtin_mma_xxsetaccz(&acc_0);

        for (i=0; i<K; i++) {
                vec_A = *((vec_t *)(a+(i*lda)));
                vec_B = *((vec_t *)(b+(i*ldb)));
                __builtin_mma_xvf32gerpp(&acc_0, vec_A, vec_B);
        }

         __builtin_mma_disassemble_acc(vec_C, &acc_0);

        *((vec_t *)(c)) = vec_C[0];
        *((vec_t *)(c+ldc)) = vec_C[1];
        *((vec_t *)(c+(2*ldc))) = vec_C[2];
        *((vec_t *)(c+(3*ldc))) = vec_C[3];

}

void sgemm(float *A, float *B, float *C, int M, int N, int K) {
    for (int i=0; i<M; i+=KM) {
    for (int j=0; j<N; j+=KN) {
        sgemm_kernel_4x4(A+i, B+j, C+j, K, M, N, N);
        }
    C += N*KM;
    }
}
```

Note: The following notes provide important programming information for successful use of the built-ins. Follow these suggestions to avoid having syntax errors reported from the front end:

- When passing arrays to built-ins that expect a void * pointer, there needs to be an explicit cast.
- When declaring vectors that are passed to built-ins, use Altivec vector syntax (such as `vector unsigned char` and `vector double`), rather than another generic vector syntax (such as `__attribute__((vector_size(16))` or similar).

Performance note: Additional care should be taken to avoid Pipeline flushes while programming MMA.

The following Pipeline flushes degrade **`gemm`** performance and should be avoided:

- A VSR Conflict Flush occurs when you access a VSR associated with a primed accumulator.
- An Accumulator Conflict Flush occurs when you perform an MMA **`ger`** instruction without priming the accumulator.

A

List of Matrix-Multiply Assist compiler built-ins

Programming note: Type `vec_t` is defined to be a normal vector unsigned char type. The `uint2`, `uint4`, and `uint8` parameters are 2-bit, 4-bit, and 8-bit unsigned integer constants, respectively. The compiler verifies that they are constants and that their values are within range.

```
void __builtin_mma_xvi4ger8 (__vector_quad *, vec_t, vec_t);
void __builtin_mma_xvi8ger4 (__vector_quad *, vec_t, vec_t);
void __builtin_mma_xvi16ger2 (__vector_quad *, vec_t, vec_t);
void __builtin_mma_xvi16ger2s (__vector_quad *, vec_t, vec_t);
void __builtin_mma_xvf16ger2 (__vector_quad *, vec_t, vec_t);
void __builtin_mma_xvbf16ger2 (__vector_quad *, vec_t, vec_t);
void __builtin_mma_xvf32ger (__vector_quad *, vec_t, vec_t);

void __builtin_mma_xvi4ger8pp (__vector_quad *, vec_t, vec_t);
void __builtin_mma_xvi8ger4pp (__vector_quad *, vec_t, vec_t);
void __builtin_mma_xvi8ger4spp(__vector_quad *, vec_t, vec_t);
void __builtin_mma_xvi16ger2pp (__vector_quad *, vec_t, vec_t);
void __builtin_mma_xvi16ger2spp (__vector_quad *, vec_t, vec_t);
void __builtin_mma_xvf16ger2pp (__vector_quad *, vec_t, vec_t);
void __builtin_mma_xvf16ger2pn (__vector_quad *, vec_t, vec_t);
void __builtin_mma_xvf16ger2np (__vector_quad *, vec_t, vec_t);
void __builtin_mma_xvf16ger2nn (__vector_quad *, vec_t, vec_t);
void __builtin_mma_xvbf16ger2pp (__vector_quad *, vec_t, vec_t);
void __builtin_mma_xvbf16ger2pn (__vector_quad *, vec_t, vec_t);
void __builtin_mma_xvbf16ger2np (__vector_quad *, vec_t, vec_t);
void __builtin_mma_xvbf16ger2nn (__vector_quad *, vec_t, vec_t);
void __builtin_mma_xvf32gerpp (__vector_quad *, vec_t, vec_t);
void __builtin_mma_xvf32gerpn (__vector_quad *, vec_t, vec_t);
void __builtin_mma_xvf32gernp (__vector_quad *, vec_t, vec_t);
void __builtin_mma_xvf32gernn (__vector_quad *, vec_t, vec_t);

void __builtin_mma_pmxvi4ger8 (__vector_quad *, vec_t, vec_t, uint4, uint4, uint8);
void __builtin_mma_pmxvi4ger8pp (__vector_quad *, vec_t, vec_t, uint4, uint4, uint8);
```

© Copyright IBM Corp. 2021.

```
void __builtin_mma_pmxvi8ger4 (__vector_quad *, vec_t, vec_t, uint4, uint4, uint4);
void __builtin_mma_pmxvi8ger4pp (__vector_quad *, vec_t, vec_t, uint4, uint4, uint4);
void __builtin_mma_pmxvi8ger4spp(__vector_quad *, vec_t, vec_t, uint4, uint4, uint4);

void __builtin_mma_pmxvi16ger2 (__vector_quad *, vec_t, vec_t, uint4, uint4, uint2);
void __builtin_mma_pmxvi16ger2s (__vector_quad *, vec_t, vec_t, uint4, uint4, uint2);
void __builtin_mma_pmxvf16ger2 (__vector_quad *, vec_t, vec_t, uint4, uint4, uint2);
void __builtin_mma_pmxvbf16ger2 (__vector_quad *, vec_t, vec_t, uint4, uint4, uint2);

void __builtin_mma_pmxvi16ger2pp (__vector_quad *, vec_t, vec_t, uint4, uint4, uint2);
void __builtin_mma_pmxvi16ger2spp (__vector_quad *, vec_t, vec_t, uint4, uint4, uint2);
void __builtin_mma_pmxvf16ger2pp (__vector_quad *, vec_t, vec_t, uint4, uint4, uint2);
void __builtin_mma_pmxvf16ger2pn (__vector_quad *, vec_t, vec_t, uint4, uint4, uint2);
void __builtin_mma_pmxvf16ger2np (__vector_quad *, vec_t, vec_t, uint4, uint4, uint2);
void __builtin_mma_pmxvf16ger2nn (__vector_quad *, vec_t, vec_t, uint4, uint4, uint2);
void __builtin_mma_pmxvbf16ger2pp (__vector_quad *, vec_t, vec_t, uint4, uint4, uint2);
void __builtin_mma_pmxvbf16ger2pn (__vector_quad *, vec_t, vec_t, uint4, uint4, uint2);
void __builtin_mma_pmxvbf16ger2np (__vector_quad *, vec_t, vec_t, uint4, uint4, uint2);
void __builtin_mma_pmxvbf16ger2nn (__vector_quad *, vec_t, vec_t, uint4, uint4, uint2);

void __builtin_mma_pmxvf32ger (__vector_quad *, vec_t, vec_t, uint4, uint4);
void __builtin_mma_pmxvf32gerpp (__vector_quad *, vec_t, vec_t, uint4, uint4);
void __builtin_mma_pmxvf32gerpn (__vector_quad *, vec_t, vec_t, uint4, uint4);
void __builtin_mma_pmxvf32gernp (__vector_quad *, vec_t, vec_t, uint4, uint4);
void __builtin_mma_pmxvf32gernn (__vector_quad *, vec_t, vec_t, uint4, uint4);

void __builtin_mma_xvf64ger (__vector_quad *, __vector_pair, vec_t);
void __builtin_mma_xvf64gerpp (__vector_quad *, __vector_pair, vec_t);
void __builtin_mma_xvf64gerpn (__vector_quad *, __vector_pair, vec_t);
void __builtin_mma_xvf64gernp (__vector_quad *, __vector_pair, vec_t);
void __builtin_mma_xvf64gernn (__vector_quad *, __vector_pair, vec_t);

void __builtin_mma_pmxvf64ger (__vector_quad *, __vector_pair, vec_t, uint4, uint2);
void __builtin_mma_pmxvf64gerpp (__vector_quad *, __vector_pair, vec_t, uint4, uint2);
void __builtin_mma_pmxvf64gerpn (__vector_quad *, __vector_pair, vec_t, uint4, uint2);
void __builtin_mma_pmxvf64gernp (__vector_quad *, __vector_pair, vec_t, uint4, uint2);
void __builtin_mma_pmxvf64gernn (__vector_quad *, __vector_pair, vec_t, uint4, uint2);

void __builtin_mma_xxmtacc (__vector_quad *);
void __builtin_mma_xxmfacc (__vector_quad *);
void __builtin_mma_xxsetaccz (__vector_quad *);

void __builtin_mma_assemble_acc (__vector_quad *, vec_t, vec_t, vec_t, vec_t);
void __builtin_mma_disassemble_acc (void *, __vector_quad *);

void __builtin_mma_assemble_pair (__vector_pair *, vec_t, vec_t);
void __builtin_mma_disassemble_pair (void *, __vector_pair *);

vec_t __builtin_xvcvspbf16 (vec_t);
vec_t __builtin_xvcvbf16sp (vec_t);
```

B

List of Matrix-Multiply Assist instructions in Power ISA v3.1

Table 5-1lists the available Matrix-Multiply Assist (MMA) instructions defined in Power ISA v3.1. For details and syntax, see Power ISA Version 3.1.

Table 5-1 MMA instructions defined in Power ISA v3.1

MMA instruction type	Traditional instructions 32-bit encoding	Prefix instructions 64-bit encoding
Data movement	**xxmfacc** **xxmtacc** **xxsetaccz**	
64-bit floating-point inputs (IEEE double-precision)	**xvf64ger2** **xvf64ger2nn** **xvf64ger2np** **xvf64ger2pn** **xvf64ger2pp**	**pmxvf64ger2** **pmxvf64ger2nn** **pmxvf64ger2np** **pmxvf64ger2pn** **pmxvf64ger2pp**
32-bit floating-point inputs (IEEE single-precision)	**xvf32ger2** **xvf32ger2nn** **xvf32ger2np** **xvf32ger2pn** **xvf32ger2pp**	**pmxvf32ger2** **pmxvf32ger2nn** **pmxvf32ger2np** **pmxvf32ger2pn** **pmxvf32ger2pp**
16-bit floating-point inputs (IEEE half-precision)	**xvf16ger2** **xvf16ger2nn** **xvf16ger2np** **xvf16ger2pn** **xvf16ger2pp**	**pmxvf16ger2** **pmxvf16ger2nn** **pmxvf16ger2np** **pmxvf16ger2pn** **pmxvf16ger2pp**
16-bit floating-point inputs (bfloat16 format)	**xvbf16ger2** **xvbf16ger2nn** **xvbf16ger2np** **xvbf16ger2pn** **xvbf16ger2pp**	**pmxvbf16ger2** **pmxvbf16ger2nn** **pmxvbf16ger2np** **pmxvbf16ger2pn** **pmxvbf16ger2pp**

© Copyright IBM Corp. 2021.

MMA instruction type	Traditional instructions 32-bit encoding	Prefix instructions 64-bit encoding
16-bit integer inputs (modulo arithmetic)	`xvi16ger2` `xvi16ger2pp`	`pmxvi16ger2` `pmxvi16ger2pp`
16-bit integer inputs (saturating arithmetic)	`xvi16ger2s` `xvi16ger2spp`	`pmxvi16ger2s` `pmxvi16ger2spp`

8-bit integer inputs (modulo/saturating)	`xvi8ger4`
	`xvi8ger4pp`
	`xvi8ger4spp`
		`pmxvi8ger4`
		`pmxvi8ger4pp`
		`pmxvi8ger4spp`

4-bit integer inputs	`xvi4ger8` `xvi4ger8pp`	`pmxvi4ger8` `pmxvi4ger8pp`

Related publications

The publications listed in this section are considered particularly suitable for a more detailed discussion of the topics covered in this paper.

Online resources

These websites are also relevant as further information sources:

- Power ISA Version 3.1

  https://wiki.raptorcs.com/w/images/f/f5/PowerISA_public.v3.1.pdf
- Anatomy of High-Performance MatrixMultiplication

  https://www.cs.utexas.edu/users/flame/pubs/GotoTOMS_final.pdf
- 754-2019 - IEEE Standard for Floating-Point Arithmetic

  https://ieeexplore.ieee.org/document/8766229
- A transprecision floating-point platform for ultra-low power computing

  https://ieeexplore.ieee.org/abstract/document/8342167

Help from IBM

IBM Support and downloads

ibm.com/support

IBM Global Services

ibm.com/services

© Copyright IBM Corp. 2021.

REDP-5612-00

ISBN 0738459453

Printed in U.S.A.

Get connected