Bit Matrix Multiplication in Commodity Processors

Yedidya Hilewitz, Cédric Lauradoux and Ruby B. Lee
Department of Electrical Engineering
Princeton University, NJ08540
hilewithz, claurado, rblee@princeton.edu

Abstract

*Registers in processors generally contain words or, with the addition of multimedia extensions, short vectors of subwords of bytes or 16-bit elements. In this paper, we view the contents of registers as vectors or matrices of individual bits. However, the facility to operate efficiently on the bit-level is generally lacking. A commodity processor usually only has logical and shift instructions and occasionally population count instructions. Perhaps the most powerful primitive bit-level operation is the bit matrix multiply (*BMM*) instruction, currently found only in supercomputers like Cray. This instruction multiplies two $n \times n$ bit matrices. In this paper, we show the power of* BMM*. We propose and analyze new processor instructions that implement simpler* BMM *primitive operations more suitable for a commodity processor. We show the impact of* BMM *on the performance of critical application kernels and discuss its hardware cost.*

1. Introduction

Many important and emerging applications could benefit from advanced bit manipulation instructions, which are not supported by current processors. These include bioinformatics, software defined radio, imaging and cryptology. Since a processor is optimized around the processing of words, it is not surprising that complex bit-level operations are typically not as well supported. While an application-specific processor may support different instructions specific to one application, a more general-purpose bit manipulation instruction is still preferred since it may be useful for new or improved algorithms, may cost less to implement than many specific instructions, and may simplify compiler support and programming.

Earlier work considered bit-level permutation instructions [13, 14, 21], bit gather and bit scatter [7] instructions or rotate and xor [3] instructions. However, all these instructions can be done by a more powerful (but more costly) instruction – bit matrix multiplication (BMM.n) which multiplies two $n \times n$ bit matrices. This is not only useful for matrix computations but also for bit manipulations on a more general scale than previous work. Moreover, the BMM instruction can also perform the functions of multimedia instructions with subword parallelism [11, 12, 6, 8] to accelerate multimedia processing.

BMM.n is currently implemented commercially only in Cray supercomputers [4, 15], for $n = 64$, as significant hardware resources are required. In this paper, we examine how smaller, lower-cost bit matrix multiply functionality can be introduced into commodity microprocessors or application-specific processors. The contributions of this paper are as follows:

- We show how BMM provides bit manipulations, multimedia subword operations and combined bit operations in addition to matrix multiplication and matrix transposition (BMMT).
- We use matrix blocking to explore optimal sub-matrix sizes for BMM primitive instructions that can more easily be deployed in commodity processors.
- We evaluate the performance of several applications and the cost tradeoffs in the design of BMM.

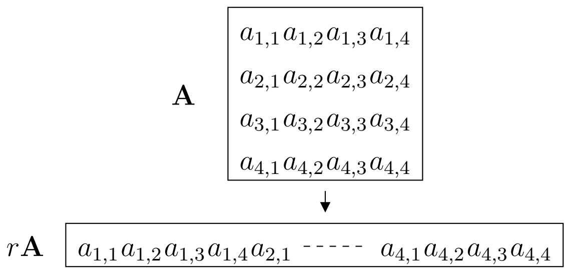
Section 2 defines BMM. Section 3 illustrates the range of operations accomplished by BMM. Section 4 shows different strategies for implementing BMM. Section 5 compares these techniques, describing performance and implementation results. Section 6 concludes the paper.

2. Definitions

The bit matrix multiply operation, BMM.n, multiplies two $n \times n$ matrices, **A** and **B**, as defined in Table 1. The rows of a matrix are numbered ascending from top to bottom, starting at 1, and the columns from left to right (Figure 1). An alternative formulation, (BMMT.n), transposes the **B** matrix. The transpose is implicit – the instruction input is **B**, but the multiplication is by $\mathbf{B}^t$.

The Cray vector supercomputers includes a BMMT.64 unit. The Cray MTA [5], a massively multithreaded (rather than vector) supercomputer has an instruction similar to

BMM.8 (with both or and xor accumulation) and an 8×8 bit matrix transpose instruction. A couple of proposed academic architectures also have a BMMT.64-like unit [25, 16].



$\mathbf{A}$
$a_{1,1}a_{1,2}a_{1,3}a_{1,4}$
$a_{2,1}a_{2,2}a_{2,3}a_{2,4}$
$a_{3,1}a_{3,2}a_{3,3}a_{3,4}$
$a_{4,1}a_{4,2}a_{4,3}a_{4,4}$
$r\mathbf{A}$
$a_{1,1}a_{1,2}a_{1,3}a_{1,4}a_{2,1}$ ----- $a_{4,1}a_{4,2}a_{4,3}a_{4,4}$

Figure 1. Data representation for BMM.4.

Table 1. BMM.n and BMMT.n.

```
BMM.n C, A, B; multiply 2 matrices A and B
C ← A × B
for i from 1 to n
  for j from 1 to n
    $c_{i,j} = a_{i,1}b_{1,j} \oplus a_{i,2}b_{2,j} \oplus \cdots \oplus a_{i,n}b_{n,j}$
```

```
BMMT.n C, A, B; multiply A and B^t
C ← A × B^T
for i from 1 to n
  for j from 1 to n
    $c_{i,j} = a_{i,1}b_{j,1} \oplus a_{i,2}b_{j,2} \oplus \cdots \oplus a_{i,n}b_{j,n}$
```

3. Capabilities of BMM

The multiplication of $n \times n$ matrices is very effective for multiplication in a finite field [20]: BMM.n is able to perform n parallel multiplications by constants in the field $\mathbf{GF}(2^n)$ in one instruction. BMM can also perform arbitrary bit permutations with or without bit repetitions, bit scatter and bit gather operations [13, 14, 7, 3].

We now show that BMM.k can also emulate subword parallel instructions useful for fast multimedia processing [11, 12]. Let us assume smaller bit matrices, with each matrix fitting into a register as shown in Figure 1, where the rows of a matrix correspond to subwords packed in a register. Different bit and subword manipulation instructions can be achieved by BMM using constant matrices denoted by $\mathcal{M}$. Then, the product $\mathbf{A} \times \mathcal{M}$ applies the same transform to each subword of register $r\mathbf{A}$, while the product $\mathcal{M} \times \mathbf{A}$ rearranges the subwords in $r\mathbf{A}$. While we show only 3×3 matrices for space reasons below, we used in BMM.k instructions, $k \times k$ matrices.

Bit or Subword permutations— An arbitrary k-bit permutation is achieved using a *permutation matrix*, *i.e.*, an $k \times k$ matrix with exactly one 1 in each row and each column and 0's elsewhere. The permutation of a k-bit vector $\mathbf{a}$ can be done as a vector-matrix product $\mathbf{a} \times \mathcal{M}$, where $\mathcal{M}$ is a permutation matrix. Using a $k \times k$ matrix $\mathbf{A}$ as input, rather than a single vector, performs k permutations in parallel on each row of the matrix $\mathbf{A}$. For example, the permutation $(1, 2, 3) \rightarrow (2, 1, 3)$ is given by $\mathbf{A} \times \mathcal{M}$:

$$\begin{pmatrix} a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,1} & a_{3,2} & a_{3,3} \end{pmatrix} \times \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} a_{1,2} & a_{1,1} & a_{1,3} \\ a_{2,2} & a_{2,1} & a_{2,3} \\ a_{3,2} & a_{3,1} & a_{3,3} \end{pmatrix}$$

If we consider the product $\mathcal{M} \times \mathbf{A}$, the permutation is applied to the columns (or over the subwords of $r\mathbf{A}$), e.g., the ALTIVEC VPERM instruction (byte permutation) can be achieved as follows:

$$\begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix} \times \begin{pmatrix} a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,1} & a_{3,2} & a_{3,3} \end{pmatrix} = \begin{pmatrix} a_{2,1} & a_{2,2} & a_{2,3} \\ a_{1,1} & a_{1,2} & a_{1,3} \\ a_{3,1} & a_{3,2} & a_{3,3} \end{pmatrix}$$

Bit gather and bit scatter — The bit gather (or parallel extract) instruction selects data bits in a register based on a mask in a second register, and right justifies the selected data bits in the destination register. The bit scatter (or parallel deposit) instruction does the reverse: it spreads right-aligned data bits in a register, according to the mask in the second register, into the destination register with zeros filling in the remaining bits [7]. Matrices can also be used to describe this type of permutation, where the zeros are treated as repeated bits. A matrix $\mathcal{M}$ is associated with a permutation with repetitions if it has at most one 1 per column and if at least one column or row is all zeroes. A given bit gather operation (deletion of column 2) over three $\mathbf{a}$ vectors is achieved by multiplying by the following $\mathcal{M}$ matrix:

$$\begin{pmatrix} a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,1} & a_{3,2} & a_{3,3} \end{pmatrix} \times \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 0 & a_{1,1} & a_{1,3} \\ 0 & a_{2,1} & a_{2,3} \\ 0 & a_{3,1} & a_{3,3} \end{pmatrix}$$

An example of a bit scatter operation (separate columns 2 and 3) is shown below:

$$\begin{pmatrix} a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,1} & a_{3,2} & a_{3,3} \end{pmatrix} \times \begin{pmatrix} 0 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} a_{1,2} & 0 & a_{1,3} \\ a_{2,2} & 0 & a_{2,3} \\ a_{3,2} & 0 & a_{3,3} \end{pmatrix}$$

An $\mathcal{M} \times \mathbf{A}$ product performs subword-parallel operations, e.g., the ALTIVEC VSPLAT instructio. Below, we show an example where the second subword (row) in $r\mathbf{A}$) is repeated:

$$\begin{pmatrix} 0 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 0 \end{pmatrix} \times \begin{pmatrix} a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,1} & a_{3,2} & a_{3,3} \end{pmatrix} = \begin{pmatrix} a_{2,1} & a_{2,2} & a_{2,3} \\ a_{2,1} & a_{2,2} & a_{2,3} \\ a_{2,1} & a_{2,2} & a_{2,3} \end{pmatrix}$$

Combined Operations — BMM.n, multiplying $\mathbf{A} \times \mathcal{M}$, can also perform multiple combined operations at once. If the matrix $\mathcal{M}$ is decomposed in the following way:

$$\mathcal{M} = \begin{pmatrix} \mathcal{M}_1 & 0 \\ 0 & \mathcal{M}_2 \end{pmatrix}$$

where $\mathcal{M}_1$ and $\mathcal{M}_2$ are two $n/2 \times n/2$ matrices, then BMM.n performs 2 BMM.$n/2$ operations on each half of each row of $\mathbf{A}$. We can have more than two matrices and their sizes do not need to be equal. BMM.n is also able to permute a subset of bits and xor with another subset of bits at the same time. We obtain a combination of permutation and a xor with a matrix defined by:

$$\mathcal{M} = \begin{pmatrix} 0 & \mathcal{M}_1 \\ 0 & \mathcal{M}_2 \end{pmatrix}.$$

If $\mathcal{M}_1$ corresponds to a circular rotation and $\mathcal{M}_2$ is the identity matrix, then the left subword of vector $\mathbf{a}$ is rotated and then xored with the right subword of $\mathbf{a}$. Thus we obtain the RORX or ROLX instruction proposed in [3], which rotates one operand and then xors it with the second. Similarly, the BMM.n instruction can compute the parity of any arbitrary subset of bits (rows or columns):

$$\begin{pmatrix} a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,1} & a_{3,2} & a_{3,3} \end{pmatrix} \times \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 0 & 0 & a_{1,2} \oplus a_{1,3} \\ 0 & 0 & a_{2,2} \oplus a_{2,3} \\ 0 & 0 & a_{3,2} \oplus a_{3,3} \end{pmatrix}$$

We have described only a few possibilities for the constant matrix $\mathcal{M}$ out of the 2^{n^2+1} possible products for BMM.

4. Implementing the BMM operation

Existing software methods — We first consider how bit matrix multiply can be implemented with existing ISAs. The best method to efficiently perform bit matrix multiplications is based on lookup tables[1]. A set of eight 256-entry tables is constructed, based on a given 64×64 $\mathbf{B}$ matrix. The $\mathbf{B}$ matrix is divided into 8 groups of 8 rows, and each table contains the xor sum of the 256 possible combinations of 8 rows $\mathbf{B}_{8i}$ to $\mathbf{B}_{8i+7}$, for $0 \leq i \leq 7$. Each byte of $r\mathbf{A}$ is then used as an index into a table and the results are xored together.

The multiplication of 64×64 matrices requires 16 KB of tables and some microprocessors only have 16KB or less of L1 data cache. If the tables are evicted from or do not fit in L1 (or higher cache levels), execution time is negatively impacted due to cache miss penalties. If a new $\mathbf{B}$ matrix is generated dynamically, the tables must be recomputed. Hence, we examine new BMM primitive operations.

New ISA — Assuming a typical processor model with 2 read operands and 1 result, implementing BMM is quite challenging. Each row of the output matrix $\mathbf{C}$ depends on a row of $\mathbf{A}$ and the entire $\mathbf{B}$ matrix. Cray supercomputers define a special 64×64 BMM register to hold the $\mathbf{B}$ matrix [4]. The contents of this register, the entire $\mathbf{B}$ matrix, is read in a single cycle. The Cray ISA defines an additional instruction to transmit the contents of a vector register to the BMM register. Subsequently, BMM instructions will stream

[1]Also known as the Four Russians algorithm [19].

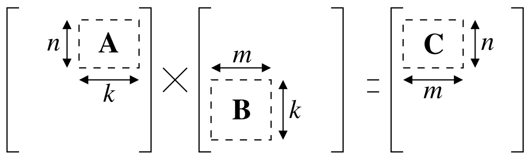


Figure 2. Decomposition into sub-matrices.

the rows of $\mathbf{A}$ from a normal vector register, multiply each row by the entire $\mathbf{B}$ matrix in a single cycle and then stream the rows of $\mathbf{C}$ back to a normal vector register.

Thus, adding BMM.64 or BMMT.64 instructions to a processor seems to require a functional unit which contains extra register storage to hold the $\mathbf{B}$ matrix. The 2 general purpose register operands hold the $\mathbf{A}$ matrix and the result $\mathbf{C}$ is written back to a general purpose register. Practically, a large unit like BMM.64 may not be desirable for a commodity or embedded processor. Thus, we investigate the use of smaller matrix sizes with matrix blocking to reduce the size of the extra storage. The main issue which must be addressed is: how to decompose the matrix multiplication ?

Matrix blocking [9] is a classical method to improve the cache usage in matrix multiplication. It is a divide and conquer method where the original matrix is divided into submatrices (Figure 2). This technique is used to perform matrix multiplication efficiently in memory constrained environments. We have used this method to find better tradeoffs, *i.e.* the choice of n, m and k, for the design of a BMM unit that performs $(n \times k) \times (k \times m)$ multiplication (Figure 2). The best parameters for the decomposition depend on four factors:

1. The number of terms computed in the submatrix multiplication: nkm. Maximizing nkm likely yields the fastest matrix multiplication when the smaller BMM instruction is used as a primitive operation.
2. The constraints coming from the processor datapath: 2-read, 1 write instructions. If we assume 64 bits registers, we have $nk \leq 128$ and $nm \leq 64$.
3. The size, mk of the extra storage.
4. The desire to perform bit or subword manipulations further constrain the choice of n, m and k to three cases:
   (a) $n = k = m$, we have a BMM.n instruction.
   (b) $n = k$ and $m = \alpha n$, we apply α different BMM.n to a given $n \times n$ matrix (product $\mathbf{A} \times \mathcal{M}$) or a single BMM.$n$ to α $n \times n$ matrices (product $\mathcal{M} \times \mathbf{A}$) if $\alpha \geq 1$. If $\alpha < 1$, we only produce αn columns.
   (c) $n = \alpha m$ and $m = k$, we apply a single BMM.m to α $m \times m$ matrices (product $\mathbf{A} \times \mathcal{M}$) or α different BMM.m to a single $m \times m$ matrices (product $\mathcal{M} \times \mathbf{A}$) if $\alpha \geq 1$. If $\alpha < 1$, we only work on αm rows.

Thus, a BMM with a square matrix or a non-square case with $\alpha > 1$ is preferable.

We have computed the best solutions with respect to the first three constraints, assuming that k is a power of two. Table 2 shows the results obtained for km= 32, 64, 128, 256 and 512 bits of storage. We observe that solutions (1)

Table 2. Best matrix sizes for $32 \leq km \leq 512$ bits of storage in the BMM functional unit.

	$nk \leq 128$	$nm \leq 64$	$km \leq 512$	nkm
(1)	16×8	16×4	8×4 (32)	512
(2)	8×16	8×4	16×4 (64)	512
(3)	**8×8**	**8×8**	**8×8 (64)**	**512**
(4)	8×16	8×8	16×8 (128)	1024
(5)	4×32	4×8	32×8 (256)	1024
(6)	4×16	4×16	16×16 (256)	1024
(7)	4×32	4×16	32×16 (512)	2048

and (2)-(4) are the most interesting if we look at the ratio of *the number of terms produced* divided by *the size of the extra storage*; they have a ratio of 16 and 8, respectively. However, the results of Table 2 do not take into account the fact that the bits in the input/output registers are sometimes not fully utilized. The output register is underused in solutions (2) and (5), but it is not clear how we can take advantage of unused output bits. Solutions (3) and (6) waste 64 bits of input, which we can use to our advantage. In solution (3), we can use the remaining 64 bits to hold the matrix **B** and, in fact, *completely remove the extra storage*. In solution (6), we can save 64 out of 256 bits of extra storage, or the extra 64 bits of input can be used to hold a partial result for accumulation.

Furthermore, solutions (3) and (6) are the only ones satisfying the last constraint. Solution (3) involves computation with square matrices (case 4(a)) which enable any of the operations described in Section 3. With solution (6), we can only perform arbitrary permutations on 4 rows (case 4(c) with $\alpha = 1/4$) since the matrix **A** is not square. Thus, solution (3) is the optimal solution in terms of cost, *i.e.*, no extra storage, full use of the registers and computational possibilities. It corresponds to the instruction BMM.8.

Table 3 shows the best parameters obtained if we assume a different datapath model, *i.e.*, (3 inputs, 1 output) or (2 inputs, 2 outputs), which is feasible with ASIPs. Solutions (8), (11) and (12) are the most interesting for bit manipulations purposes. Solution (8) can perform 8 BMM.4 operations, solution (11) 2 BMM.8, and solution (12) a BMM.16 on 8 rows. However, requiring 2 writes is significantly more expensive, so we further evaluate only (3) and (6) from Table 2. In the rest of the paper, we denote the solution (3) by BMM.8 and the solution (6) by BMM.16.

Table 3. Assuming non-standard datapaths in application-specific processors.

	2 reads - 2 writes			
	nk	nm	km	nkm
(8)	32×4	32×4	4×4 (16)	512
(9)	16×8	16×4	8×4 (32)	512
(10)	16×4	16×8	4×8 (32)	512
(11)	16×8	16×8	8×8 (64)	1024
(12)	8×16	8×16	16×16 (256)	2048
	3 reads - 1 writes			
(13)	12×16	12×4	16×4 (64)	768
(14)	6×32	6×8	32×8 (256)	1536

To achieve multiplication of larger bit matrices with blocking involves resizing submatrices, to get data laid out as desired. Resizing bit matrices is equivalent to transposing a 2-D matrix of subwords. For 64×64 matrices, a 1×64-bit block is equivalent to 8 byte elements from the same row of eight 8×8 byte matrices or vice versa. The MIX instruction can be used to perform matrix transpose efficiently as shown in [12], e.g., eight 8×8 matrices can be transposed in parallel using $8 \times \log_2(8) = 24$ MIX instructions. Hence, resizing 64 (1×64) blocks to 64 (8×8) blocks can be done with $8 \times 24 = 192$ MIX instructions.

5. Applications, Performance and Cost

5.1. Applications

We have tested our instructions with applications involving bit manipulation, finite field multiplication and large matrix multiplication. More information on the following kernels can be found in [17] and [24].

LFSR (LFSR) — Linear Feedback Shift Registers are the underpinning of stream ciphers, spread spectrum systems and FEC computations (CRC and convolutional codes). We have used a 64-bit LFSR, representing the computation involved in an LFSR by a sequence of permute and xor instructions (see Section 3 - Combined operations). For an m-bit LFSR, we need $\frac{m}{n} + 2$ BMM.n operations. Our use of BMM is based on the bitslicing technique in [10].

FFT (FFT) — The Fast Fourier Transform is a fundamental building block used in DSP systems, with applications ranging from OFDM systems to image processing algorithms. We measure the bit manipulation capabilities of BMM.k in FFT.

Finite Field (GF) — Multiplication by a constant in the finite field $GF(256)$ is critical in Reed-Solomon encoders, stream ciphers and block ciphers. We have used a lookup table technique as a baseline and the matrix representation of [20] for the BMM implementation.

Bioinformatics DNA Reverse-Complement (REVC) — DNA strings consist of the four bases A, C, G and T, which can be represented with 2-bit symbols in compressed form. To obtain the reverse-complement string, the sequence of 2-bit symbols is reversed and the first bit is complemented. Reversing the symbols measures the bit manipulation capabilities of BMM.

Bitslice Logic (BSBOX) — Bitslicing is a very efficient way to evaluate logic for simulation [18]. This kernel measures the processor's ability to change the representation of Boolean matrices. We used a bitslice SBOX of DES [2] as simulated logic. The critical part of this kernel is the 2 transpositions of a 64×64-bit matrix. The purpose of this kernel is to determine the usefulness of the BMMT.n instruction.

The kernels **LFSR**, **FFT** and **GF** are typical DSP algorithms. Some DSPs have special-purpose instructions for these kernels, e.g., the TMS320 DSP [23] has instructions to perform finite field multiplications (GMPY4) and bit reversal (BITR). However, they have a dedicated instruction for each kernel. We propose a single instruction to enhance all of them. The kernels **BSBOX** and **REVC** are more related to computational simulations found in supercomputing.

5.2. Performance

We coded the benchmarks and simulated them using the SimpleScalar Alpha simulator [1] to test the performance of BMM.8, BMMT.8, BMM.16, BMMT.16, BMM.64 and BMMT.64 compared to the fastest software implementation. We simulated with a 4-way superscalar processor with 1 BMM unit.

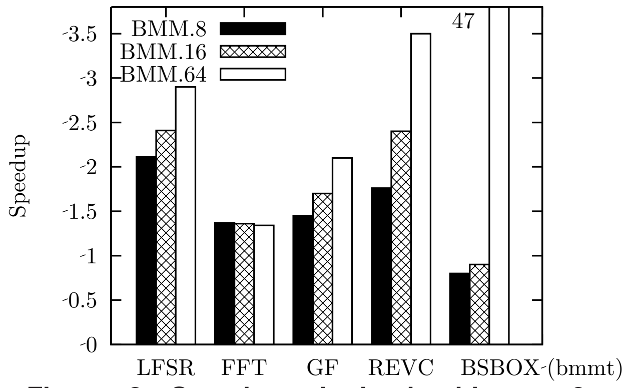


Figure 3. Speedup obtained with BMM.8, BMM.16 and BMM.64.

Figure 3 shows the results. **LFSR**, **GF**, **FFT** and **REVC** emphasize the bit manipulations and multiplication capabilities of BMM.8 , BMM.16 and BMM.64. On these four kernels, BMM.8 shows a 1.7× speedup on average for these applications while BMM.16 and BMM.64 show respectively a 1.9× and 2.3× speedup. BMM.64 is able to perform 64 operations on a byte as opposed to only 8 with BMM.8. However, the parallelism provided by BMM.64 is difficult to exploit as the other operations in the benchmarks do not exhibit as much parallelism. This explains the result obtained for the **FFT** benchmark. Another issue for BMM.64 is the load/store of the two 64×64 matrices which significantly impact the performance. For **BSBOX**, which stresses transpose and is the only kernel using the BMMT.k variants, BMMT.8 and BMMT.16 show a *slowdown*. This can be attributed to the overhead of resizing. BMMT.64 shows tremendous speedup, 47×, since no resizing is needed. Hence, BMM.8 and BMM.16 are efficient compared to the much larger BMM.64, but BMMT.8 and BMMT.16 may be less useful compared to BMMT.64.

5.3. Cost

We now consider the implementation costs of the different BMM.k instructions. Table 4 lists the latencies and areas of the new functional units. An example of the layout of BMMT.4 unit is given in Figure 4. We synthesized the various new functional units mapping to a TSMC 90 nm standard cell library [22]. A reference ALU was synthesized and the other designs were constrained to match the latency of the ALU. The BMM.64 functional unit has slightly greater latency than the ALU. Thus it might take 1 or 2 cycles depending on timing constraints. BMM.64 is also much larger than an ALU (around 6.3×). The basic BMM.8 is faster than an ALU and less than a quarter of the size. BMM.16 is just as fast as an ALU and 84% the size.

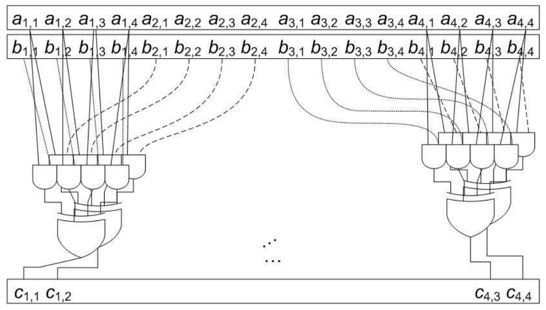

$a_{1,1}$ $a_{1,2}$ $a_{1,3}$ $a_{1,4}$ $a_{2,1}$ $a_{2,2}$ $a_{2,3}$ $a_{2,4}$ $a_{3,1}$ $a_{3,2}$ $a_{3,3}$ $a_{3,4}$ $a_{4,1}$ $a_{4,2}$ $a_{4,3}$ $a_{4,4}$
$b_{1,1}$ $b_{1,2}$ $b_{1,3}$ $b_{1,4}$ $b_{2,1}$ $b_{2,2}$ $b_{2,3}$ $b_{2,4}$ $b_{3,1}$ $b_{3,2}$ $b_{3,3}$ $b_{3,4}$ $b_{4,1}$ $b_{4,2}$ $b_{4,3}$ $b_{4,4}$
$c_{1,1}$ $c_{1,2}$
$c_{4,3}$ $c_{4,4}$

Figure 4. Layout of BMMT.4 unit.

When comparing BMM.16 and BMM.64 to BMM.8, BMM.16 has 3.6× the area for only 1.15× the performance for the benchmarks considered and BMM.64 has 26.9× the area, while only achieving 1.4× the performance. Clearly, BMM.16 and BMM.64 achieve better performance, but at a steep cost.

6. Conclusion

In this paper, we have discussed how commodity processors can increase performance for bit matrix multiply operations. We define a set of primitive BMM instructions that multiply submatrices of **A** and **B**. We consider instructions that either use GPRs or auxiliary storage to hold the **B** submatrix. In particular, we focus on the BMM.8 instruction that does not require any extra storage. This instruction is comparable to or faster than the table lookup algorithm, the current best method for bit matrix multiplication. We have

Table 4. Latency and area of BMM functional units performing square-matrix bmm computations.

	Extra storage	Latency (ns)	Latency (relative to ALU)	Cycles	Area (NAND gates)	Area (relative to ALU)
ALU	-	0.500	1	1	6.4K	1
Table Lookup	16 KB tables	-	-	cache latency	-	-
BMM.8	-	0.371	0.74	1	1.5K	0.23
BMM.16	256 bits	0.5	1	1	5.4K	0.84
BMM.64 (∼Cray)	4096 bits	0.513	1.03	1 or 2	40.4K	6.31

validated the performance using a variety of benchmark kernels. Our results show that BMM.8 exhibited 1.7× speedup on average over the baseline case, BMM.16 1.9× speedup and BMM.64 2.3× speedup. Overall, BMM.64 was 1.4× faster than BMM.8 on average, but there were cases where it was no better and other cases where it was much better.

We have also looked at the implementation cost of the new instructions. BMM.8 is smaller than an ALU and is a fraction of the size of the full BMM.64 as implemented by Cray. Thus BMM.8 is a good candidate for inclusion in small processors and might be suitable for inclusion within the ALU itself.

Clearly to achieve the best performance, a full Cray-like BMM.64 unit can be implemented. However, the BMM.8 instruction requires no extra state and can, by itself, perform many of the existing SIMD ISA subword manipulation instructions. Furthermore, for some applications, it achieved the same performance as BMM.64. Overall, our proposed instructions are a significant improvement over the current software methods for computing bit matrix multiply and narrow the gap between supercomputer and commodity processor performance, at considerable cost savings.

References

[1] T. M. Austin, E. Larson, and D. Ernst. Simplescalar: An infrastructure for computer system modeling. *IEEE Computer*, 35(2):59–67, 2002.

[2] E. Biham. A Fast New DES Implementation in Software. In *Fast Software Encryption - FSE '97*, pages 260–272, 1997.

[3] J. Burke, J. McDonald, and T. M. Austin. Architectural Support for Fast Symmetric-Key Cryptography. In *Architectural Support for Programming Languages and Operating Systems, ASPLOS 2000*, pages 178–189. ACM, 2000.

[4] Cray Inc. Cray Assembly Language (CAL) for Cray X1 Systems Reference Manual, version 1.2. Cray Inc., 2003.

[5] Cray, Inc. CRAY/MTA Principles of Operation, 2005.

[6] K. Diefendorff, P. K. Dubey, R. Hochsprung, and H. Scales. AltiVec Extension to PowerPC Accelerates Media Processing. *IEEE Micro*, 20(2):85–95, 2000.

[7] Y. Hilewitz and R. B. Lee. Fast Bit Compression and Expansion with Parallel Extract and Parallel Deposit Instructions. In *IEEE International Conference on Application-Specific Systems, Architecture and Processors*, pages 65–72, 2006.

[8] Intel. Intel 64 and IA-32 Architectures Software Developers Manual Volume 2b: Instruction Set Reference, N-Z. Intel Corporation, 2006.

[9] M. Kowarschik and C. Weiß. An Overview of Cache Optimization Techniques and Cache-Aware Numerical Algorithms. In *Algorithms for Memory Hierarchies*, Lecture Notes in Computer Science 2625, pages 213–232, 2002.

[10] C. Lauradoux. From Hardware to Software Synthesis of Linear Feedback Shift Registers. In *Proceedings of Workshopon Performance Optimization for High-Level Languages and Libraries - POHLL 2007*, Mar. 2007.

[11] R. B. Lee. Accelerating Multimedia with Enhanced Microprocessors. *IEEE Micro*, 15(2):22–32, 1995.

[12] R. B. Lee. Subword parallelism with MAX-2. *IEEE Micro*, 16(4):51–59, 1996.

[13] R. B. Lee, Z. Shi, and X. Yang. Efficient permutation instructions for fast software cryptography. *IEEE Micro*, 21(6):56–69, 2001.

[14] R. B. Lee, Z. Shi, and X. Yang. How a Processor can Permute n bits in O(1) cycles. In *Proceedings of Hot Chips 14*, Aug. 2002.

[15] W. Lee, G. J. Geissler, S. J. Johnson, and A. J. Schiffleger. Vector Bit-Matrix Multiply Function Unit, 1996.

[16] C. Lemuet, J. Sampson, J.-F. Collard, and N. Jouppi. The potential energy efficiency of vector acceleration. In *ACM/IEEE conference on Supercomputing - SC '06*, page 77. ACM, 2006.

[17] S. Lin and D. J. Costello. *Error Control Coding, Second Edition*. Prentice-Hall, Inc., 2004.

[18] L.Soulé. *Parallel Logic Simulation: An Evaluation of Centralized-Time and Distributed-Time Algorithms*. PhD thesis, Stanford University, 1992.

[19] U. Manber. *Introduction to Algorithms: A Creative Approach*. Addison-Wesley, 1989.

[20] C. Paar. *Efficient VLSI Architectures for Bit Parallel Computation in Galois Fields*. PhD thesis, Universität GH Essen, 1994.

[21] Z. Shi, X. Yang, and R. B. Lee. Arbitrary Bit Permutations in One or Two Cycles. In *Proceedings of the IEEE International Conference on Application-Specific Systems, Architectures and Processors (ASAP)*, pages 237–247, June 2003.

[22] Taiwan Semiconductor Manufacturing Corporation. TCBN90GTHP: TSMC 90nm Core Library Databook, version 1.1, 2006.

[23] Texas Instrument. TMS320C64x Technical Overview, 2001.

[24] H. C. van Tilborg. *Encyclopedia of Cryptography and Security*. Springer-Verlag, 2005.

[25] M. Wei, M. Snir, J. Torrellas, and R. B. Tremaine. A Near-Memory Processor for Vector, Streaming and Bit Manipulation Workloads. In *Watson Conference on Interaction between Architecture, Circuits, and Compilers*, 2005.