Matrix Multiplication over $\mathbb{F}_2$ in the M4RI library

Martin Albrecht
(joint work with Gregory Bard and Bill Hart)

SD10, Nancy, October 10, 2008
Outline

Multiplication

Loops, Cache & SSE2

Multi-Core

Final
Outline

Multiplication

Loops, Cache & SSE2

Multi-Core

Final
Field with two elements.

- Logical bitwise XOR is addition.

- Logical bitwise AND is multiplication.

- 64 (128) basic operations in at most one CPU cycle

- Arithmetic rather cheap

<table>
<thead>
<tr>
<th></th>
<th>⊕</th>
<th>⊙</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0 1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1 0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1 1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
Why Bother?

Matrix multiplication

- is the fundamental building block for other linear algebra operations,
- is for examples used in PolyBoRi in Gröbner basis calculations,
- ... is fun.

$\mathbb{F}_2$

- is extensively used all over the place,
- is quite different from $\mathbb{F}_p$ for $p > 2$ prime,
- ... is also fun.
Consider $C = A \cdot B$ ($A$ is an $m \times l$ matrix, $B$ is an $l \times n$ matrix).

$A$ can be divided into $l/k$ vertical “stripes” $A_0 \ldots A_{(l-1)/k}$ of $k$ columns each.

$B$ can be divided into $l/k$ horizontal “stripes” $B_0 \ldots B_{(l-1)/k}$ of $k$ rows each.

For simplicity assume $k$ divides $l$.

We have:

$$C = A \cdot B = \sum_{0}^{(l-1)/k} A_i \cdot B_i.$$
M4RM [1] II

\[ A = \begin{pmatrix} 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \end{pmatrix}, \quad B = \begin{pmatrix} 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{pmatrix}, \quad A_0 = \begin{pmatrix} 1 & 1 \\ 0 & 0 \\ 1 & 1 \\ 0 & 1 \end{pmatrix} \]

\[ A_1 = \begin{pmatrix} 0 & 1 \\ 0 & 0 \\ 1 & 1 \\ 1 & 1 \end{pmatrix}, \quad B_0 = \begin{pmatrix} 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 \end{pmatrix}, \quad B_1 = \begin{pmatrix} 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{pmatrix} \]

\[ A_0 \cdot B_0 = \begin{pmatrix} 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{pmatrix}, \quad A_1 \cdot B_1 = \begin{pmatrix} 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 \end{pmatrix} \]
M4RM: Gray Codes

0 0 0
0 0 1
0 1 1
0 1 0
1 1 0
1 1 1
1 0 1
1 0 0

► Computing all possible $2^k - 1$ sums, costs only $2^k - 1$ additions.
M4RM: Algorithm

def add_row_from_table(C, r, T, x):
    for 0 <= i < C.ncols():
        C[r,i] += T[x,i]

def m4rm(A, B, k):
    m = A.nrows(); l = A.ncols(); n = B.ncols()
    C = Matrix(GF(2), m, n)
    for 0 <= i < l/k:
        T = make_table(B, i*k, 0, k)
        for 0 <= j < m:
            x = read_bits(A, j, k*i, k)
            add_row_from_table(C, j, T, x)
    return C

- Fastest known practical algorithm is Strassen-Winograd multiplication ($\mathcal{O}(n^{\log_2 7})$)
- M4RM can be used as base case for small dimensions
- Optimisation of this base case crucial for competitive performance

All timings in this talk are for Strassen-Winograd.
Outline

Multiplication

Loops, Cache & SSE2

Multi-Core

Final
XOR is Cheap, Loops are Expensive

\[
\text{for } (i = 0; i < 2048; i++) \{
\text{dst}[i] ^= \text{src}[i];
\}
\]

Don’t take this example too seriously, your compiler is your friend, don’t try to outsmart it: It will outsmart you and unroll loops on the way.
Modern compilers (GCC 4, MSVC, SunCC) support 128-bit SSE2 integer instructions via compiler intrinsics.

```c
while (__c < eof) {
    xmm1 = _mm_xor_si128(*__c, *__t0++);
    xmm1 = _mm_xor_si128(*__c, *__t1++);
    xmm1 = _mm_xor_si128(*__c, *__t2++);
    xmm1 = _mm_xor_si128(*__c, *__t3++);
    *__c++ = xmm1;
}
```
### The SSE2 Instruction Set II

<table>
<thead>
<tr>
<th>Matrix Dimensions</th>
<th>Using 64-bit</th>
<th>Using 128-bit (SSE2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10,000 × 10,000</td>
<td>1.981</td>
<td>1.504</td>
</tr>
<tr>
<td>16,384 × 16,384</td>
<td>7,906</td>
<td>6.074</td>
</tr>
<tr>
<td>20,000 × 20,000</td>
<td>14.076</td>
<td>10.721</td>
</tr>
<tr>
<td>32,000 × 32,000</td>
<td>56.931</td>
<td>43.197</td>
</tr>
</tbody>
</table>

Table: Strassen-Winograd multiplication on 64-bit Linux, 2.33Ghz Core 2 Duo
Cache [3] 1

<table>
<thead>
<tr>
<th>Memory</th>
<th>Regs</th>
<th>L1</th>
<th>L2</th>
<th>Ram</th>
<th>Swap</th>
</tr>
</thead>
<tbody>
<tr>
<td>Speed (ns)</td>
<td>0.5</td>
<td>2</td>
<td>6</td>
<td>$10^2$</td>
<td>$10^7$</td>
</tr>
<tr>
<td>Cost (cycles)</td>
<td>1</td>
<td>4</td>
<td>14</td>
<td>200</td>
<td>$2 \cdot 10^7$</td>
</tr>
<tr>
<td>Size</td>
<td>$4 \cdot 64$-bit</td>
<td>64k</td>
<td>1-4M</td>
<td>1G</td>
<td>100G</td>
</tr>
</tbody>
</table>
“Therefore, we propose that matrix entry reads and writes be tabulated, because addition (XOR) and multiplication (AND) are single instructions, while reads and writes on rectangular arrays are much more expensive. Clearly these data structures are nontrivial in size (hundreds of megabytes at the least) and so memory transactions will be the bulk of the computational burden.”

— Gregory Bard, [2]
Assume that $A$ and $C$ do not fit into L2 cache.

```python
def m4rm(A, B, k):
    m = A.nrows(); l = A.ncols(); n = B.ncols()
    C = Matrix(GF(2), m, n)

    for 0 <= i < l//k:
        # this is cheap
        T = make_table(B, i*k, 0, k)
        for 0 <= j < m:
            # we touch each row of $A$ and $C$ only once
            x = read_bits(A, j, k*i, k)
            add_row_from_table(C, j, T, x)

    return C
```
def m4rm_cf(A, B, k):
    m = A.nrows(); l = A.ncols(); n = B.ncols()
    C = Matrix(GF(2), m, n)

    for 0 <= start < m/block_size:
        for 0 <= i < l/k:
            T = make_table(B, i*k, 0, k)
            # we don't wander off beyond block_size
            for 0 <= s < block_size:
                j = start*block_size + s;
                x = read_bits(A, j, k*i, k)
                add_row_from_table(C, j, T, x)
    return C
### Cache Friendly M4RM III

<table>
<thead>
<tr>
<th>Matrix Dimensions</th>
<th>Plain</th>
<th>Cache Friendly</th>
</tr>
</thead>
<tbody>
<tr>
<td>10,000 × 10,000</td>
<td>4.141</td>
<td>2.866</td>
</tr>
<tr>
<td>16,384 × 16,384</td>
<td>16.434</td>
<td>12.214</td>
</tr>
<tr>
<td>20,000 × 20,000</td>
<td>29.520</td>
<td>20.497</td>
</tr>
<tr>
<td>32,000 × 32,000</td>
<td>86.153</td>
<td>82.446</td>
</tr>
</tbody>
</table>

Table: Strassen-Winograd with different base cases on 64-bit Linux, 2.33Ghz Core 2 Duo
actual arithmetic is quite cheap compared to memory reads and writes

the cost of memory accesses greatly depends on where in memory data is located

try to fill all of L1 with Gray code tables.

Example: $k = 10$ and 1 Gray code table $\rightarrow$ 10 bits at a time. $k = 9$ and 2 Gray code tables, still the same memory for the tables but deal with 18 bits at once.

The price is one extra row addition, which is cheap if the operands are all in cache.
def m4rm_2t(A, B, k):
    m = A.nrows(); l = A.ncols(); n = B.ncols()
    C = Matrix(GF(2), m, n)
    for 0 <= i < l/(2*k):
        T0 = make_table(B, 2*i*k, 0, k)
        T1 = make_table(B, 2*i*k + k, 0, k)
        for 0 <= j < m:
            x0 = read_bits(A, j, 2*k*i, k)
            x1 = read_bits(A, j, 2*k*i+k, k)
            add_2rows_from_table(C, j, T0, x0, T1, x1)
    return C
### Table: Strassen-Winograd with different base cases on 64-bit Linux, 2.33Ghz Core 2 Duo

<table>
<thead>
<tr>
<th>Matrix Dimensions</th>
<th>$t = 1$</th>
<th>$t = 2$</th>
<th>$t = 8$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$10,000 \times 10,000$</td>
<td>4.141</td>
<td>1.982</td>
<td>1.599</td>
</tr>
<tr>
<td>$16,384 \times 16,384$</td>
<td>16.434</td>
<td>7.258</td>
<td>6.034</td>
</tr>
<tr>
<td>$20,000 \times 20,000$</td>
<td>29.520</td>
<td>14.655</td>
<td>11.655</td>
</tr>
<tr>
<td>$32,000 \times 32,000$</td>
<td>86.153</td>
<td>49.768</td>
<td>44.999</td>
</tr>
</tbody>
</table>
Parameter Choices

cutoff two matrices fit into L2 cache

blocksize reduces the size of the matrices we are working with to actually fit three matrices in L2 cache.

\[ k \text{ is either } \lfloor 0.75 \cdot \log_2 \text{blocksize} \rfloor - 2 \text{ or } \lfloor 0.75 \cdot \log_2 \text{blocksize} \rfloor - 3 \text{ depending on the input dimensions and the size of the L1 cache.} \]

Opteron: \( \text{cutoff} = 2048, \text{blocksize} = 1024, k = 5, t = 8 \)
Core 2 Duo: \( \text{cutoff} = 4096, \text{blocksize} = 2048, k = 6, t = 8 \)
Results: Multiplication I

Matrix Multiplication, Debian/GNU Linux, 64-Bit, Opteron

- Magma time
- GAP time
- M4RI time

Figure: 2.6 Ghz Opteron, 18GB RAM
Results: Multiplication II

Matrix Multiplication, Debian/GNU Linux, 64-Bit, Core 2 Duo

Figure: 2.33 Ghz Core 2 Duo, 3GB RAM
Results: Multiplication III

Matrix Multiplication for Random Matrices over $F_2$: Sage vs Magma

Figure: 2.33 Ghz Core 2 Duo, 3GB RAM
Results: Multiplication IV

Figure: 2.33 Ghz Core 2 Duo, 3GB RAM
Outline

Multiplication

Loops, Cache & SSE2

Multi-Core

Final
Parallelisation I

- Strassen-Winograd is less suitable for parallel computing than Strassen
- Strassen is less suitable for parallel computing than cubic multiplication

Strategy
Use parallel cubic multiplication until all cores are utilised, then Strassen-Winograd or M4RM on each core depending on submatrix dimensions.
Parallelisation II

```
#pragma omp parallel sections
{
#pragma omp section
{
    _mzd_mul(Q0, A00, B00, cut);
}
#pragma omp section
{
    _mzd_mul(Q1, A01, B10, cut);
}
}

“The OpenMP API supports multi-platform shared-memory parallel programming in C/C++ ... It is a portable, scalable model ... on platforms from the desktop to the supercomputer.”
```
Results: OpenMP I

Figure: 2.33 Ghz Core 2 Duo, 3GB RAM, L2 shared
Results: OpenMP II

Matrix Multiplication, Debian/GNU Linux, 64-Bit, Core 2 Quad

- 1 core
- 2 cores, L2 shared
- 2 cores, L2 dedicated
- 4 cores

Figure: 2.4 Ghz Core 2 Quad, 8GB RAM (eno)
Results: OpenMP III

![Graph showing speed-up for matrix multiplication with different core configurations.]

Figure: 2.4 Ghz Core 2 Quad, 8GB RAM (eno)
Results: OpenMP IV

Matrix Multiplication, Debian/GNU Linux, 64-Bit, Opteron

- 1 core
- 2 cores
- 4 cores

Figure: 2.6 Ghz Opteron, 18GB RAM, **L2 not shared**
Results: OpenMP V

Figure: 2.6 Ghz Opteron, 18GB RAM, **L2 not shared**
Thank You!
