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

$\mathbb{F}_2$

- 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

		$\oplus$	$\odot$
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

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.

M4RM [1] I

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" $A_0 \dots A_{(l-1)/k}$ of k columns each.

B can be divided into l/k horizontal "stripes" $B_0 \dots B_{(l-1)/k}$ 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

$$A = \begin{pmatrix} 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \end{pmatrix}, B = \begin{pmatrix} 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{pmatrix}, A_0 = \begin{pmatrix} \mathbf{1} & \mathbf{1} \\ 0 & 0 \\ \mathbf{1} & \mathbf{1} \\ 0 & 1 \end{pmatrix}$$

$$A_1 = \begin{pmatrix} 0 & 1 \\ 0 & 0 \\ \mathbf{1} & \mathbf{1} \\ \mathbf{1} & \mathbf{1} \end{pmatrix}, B_0 = \begin{pmatrix} 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 \end{pmatrix}, B_1 = \begin{pmatrix} 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{pmatrix}$$

$$A_0 \cdot B_0 = \begin{pmatrix} \mathbf{1} & \mathbf{1} & \mathbf{0} & \mathbf{1} \\ 0 & 0 & 0 & 0 \\ \mathbf{1} & \mathbf{1} & \mathbf{0} & \mathbf{1} \\ 0 & 1 & 1 & 0 \end{pmatrix}, A_1 \cdot B_1 = \begin{pmatrix} 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{1} \\ \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{1} \end{pmatrix}$$

M4RM: Gray Codes

0	0
0	**1**
1	1
1	**0**

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
```

Strassen-Winograd [4] Multiplication

- Fastest known pratical algorithm is Strassen-Winograd multiplication ($\mathcal{O}\left(n^{\log_2 7}\right)$)
- 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.

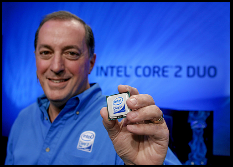
Outline

Multiplication

Loops, Cache & SSE2

Multi-Core

Final



XOR is Cheap, Loops are Expensive

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

```
400567:  488b04d3        mov   (%rbx,%rdx,8),%rax
40056b:  483144d500      xor   %rax,0x0(%rbp,%rdx,8)
400570:  4883c201        add   $0x1,%rdx
400574:  4881fa00080000  cmp   $0x800,%rdx
40057b:  75ea            jne   400567 <main+0x2f>
```

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.

The SSE2 Instruction Set I

Modern compilers (GCC 4, MSVC, SunCC) support 128-bit SSE2 integer instructions via compiler intrinsics.

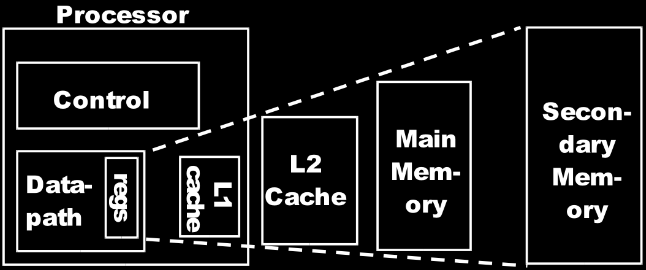
```
while( __c < eof) {
  xmm1 = _mm_xor_si128(*__c, *__t0++);
  xmm1 = _mm_xor_si128(*__c, *__t1++);
  xmm1 = _mm_xor_si128(*__c, *__t2++);
  xmm1 = _mm_xor_si128(*__c, *__t3++);
  *__c++ = xmm1;
}
```

The SSE2 Instruction Set II

Matrix Dimensions	Using 64-bit	Using 128-bit (SSE2)
$10,000 \times 10,000$	1.981	1.504
$16,384 \times 16,384$	7,906	6.074
$20,000 \times 20,000$	14.076	10.721
$32,000 \times 32,000$	56.931	43.197

Table: Strassen-Winograd multiplication on 64-bit Linux, 2.33Ghz Core 2 Duo

Cache [3] I



Memory	Regs	L1	L2	Ram	Swap
Speed (ns)	0.5	2	6	10^2	10^7
Cost (cycles)	1	4	14	200	$2 \cdot 10^7$
Size	$4 \cdot$ 64-bit	64k	1-4M	1G	100G

Cache [3] II

> *"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]

Cache Friendly M4RM I

Assume that A and C do not fit into L2 cache.

```
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
```

Cache Friendly M4RM II

```
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

Matrix Dimensions	Plain	Cache Friendly
$10,000 \times 10,000$	4.141	2.866
$16,384 \times 16,384$	16.434	12.214
$20,000 \times 20,000$	29.520	20.497
$32,000 \times 32,000$	86.153	82.446

Table: Strassen-Winograd with different base cases on 64-bit Linux, 2.33Ghz Core 2 Duo

$t > 1$ Gray Code Tables I

- 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.

$t > 1$ Gray Code Tables II

```
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
```

$t > 1$ Gray Code Tables III

Matrix Dimensions	$t = 1$	$t = 2$	$t = 8$
$10,000 \times 10,000$	4.141	1.982	1.599
$16,384 \times 16,384$	16.434	7.258	6.034
$20,000 \times 20,000$	29.520	14.655	11.655
$32,000 \times 32,000$	86.153	49.768	44.999

Table: Strassen-Winograd with different base cases on 64-bit Linux, 2.33Ghz Core 2 Duo

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
: is either $\lfloor 0.75 \cdot \log_2 \mathit{blocksize} \rfloor - 2$ or $\lfloor 0.75 \cdot \log_2 \mathit{blocksize} \rfloor - 3$ depending on the input dimensions and the size of the L1 cache.

Opteron: $\mathit{cutoff} = 2048$, $\mathit{blocksize} = 1024$, $k = 5$, $t = 8$
Core 2 Duo: $\mathit{cutoff} = 4096$, $\mathit{blocksize} = 2048$, $k = 6$, $t = 8$

Results: Multiplication I

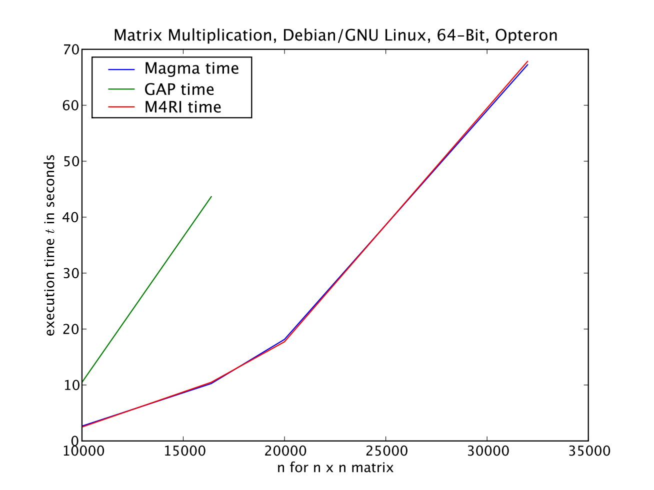


Figure: 2.6 Ghz Opteron, 18GB RAM

Results: Multiplication II

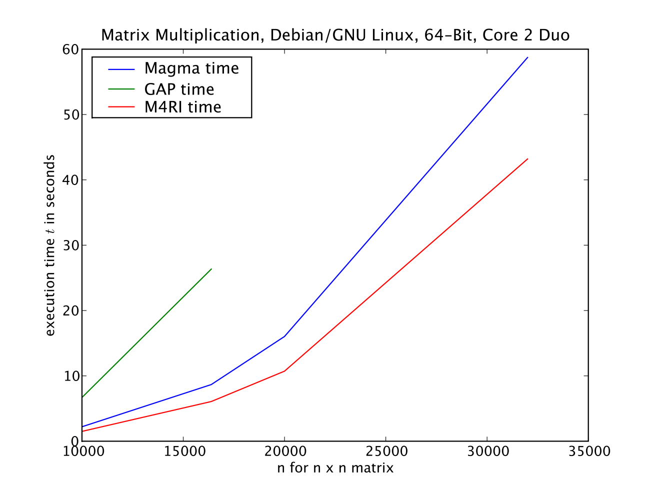


Figure: 2.33 Ghz Core 2 Duo, 3GB RAM

Results: Multiplication III

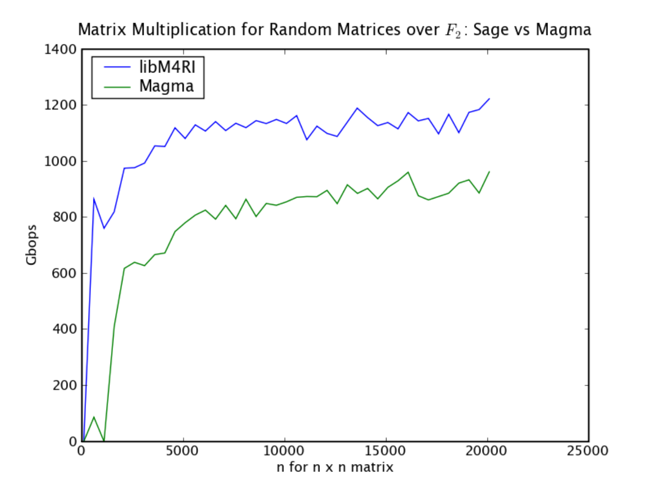


Figure: 2.33 Ghz Core 2 Duo, 3GB RAM

Results: Multiplication IV

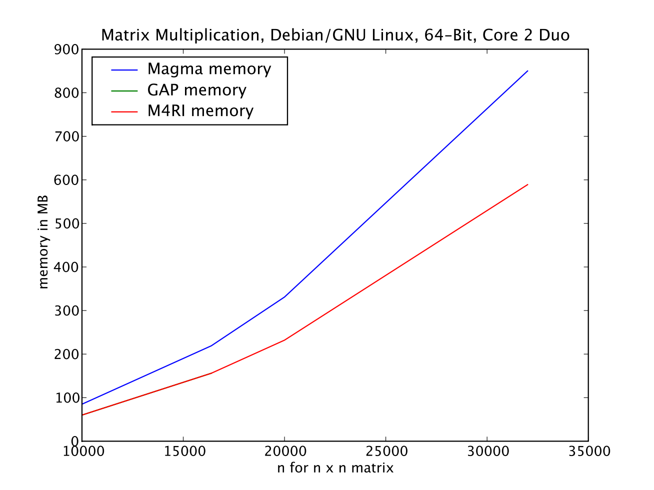


Figure: 2.33 Ghz Core 2 Duo, 3GB RAM

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

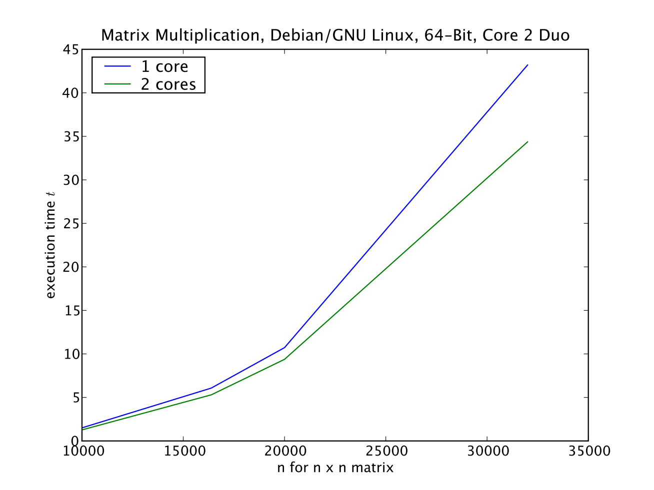


Figure: 2.33 Ghz Core 2 Duo, 3GB RAM, **L2 shared**

Results: OpenMP II

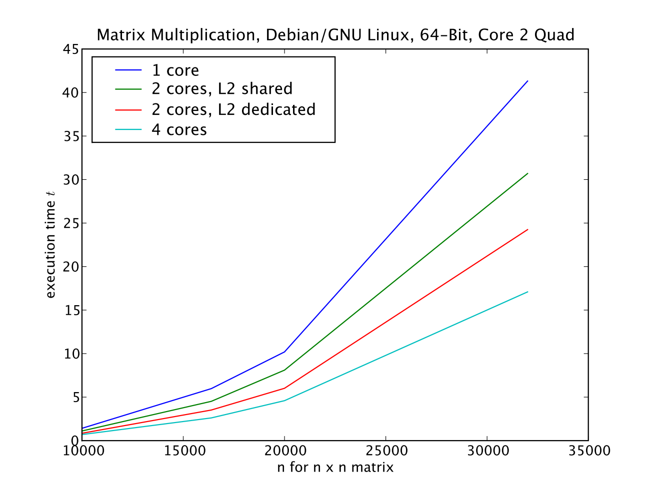


Figure: 2.4 Ghz Core 2 Quad, 8GB RAM (eno)

Results: OpenMP III

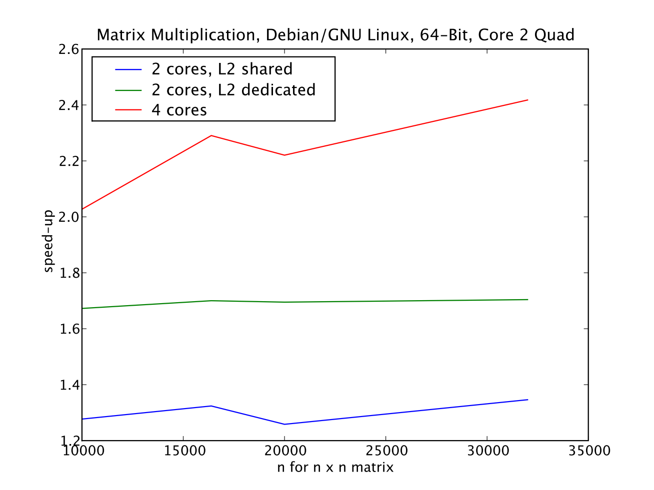


Figure: 2.4 Ghz Core 2 Quad, 8GB RAM (eno)

Results: OpenMP IV

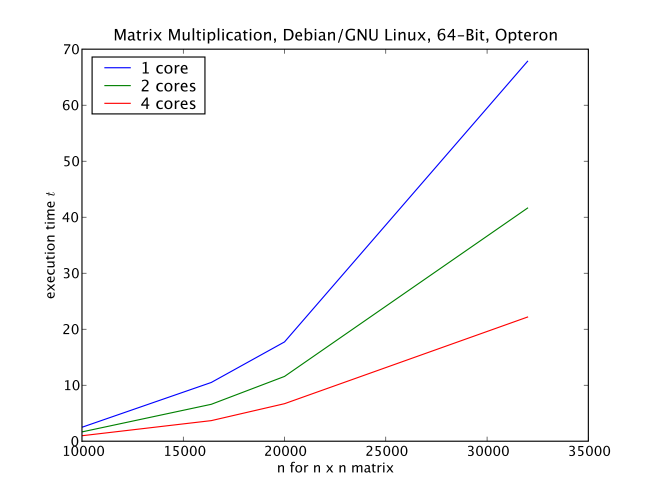


Figure: 2.6 Ghz Opteron, 18GB RAM, **L2 not shared**

Results: OpenMP V

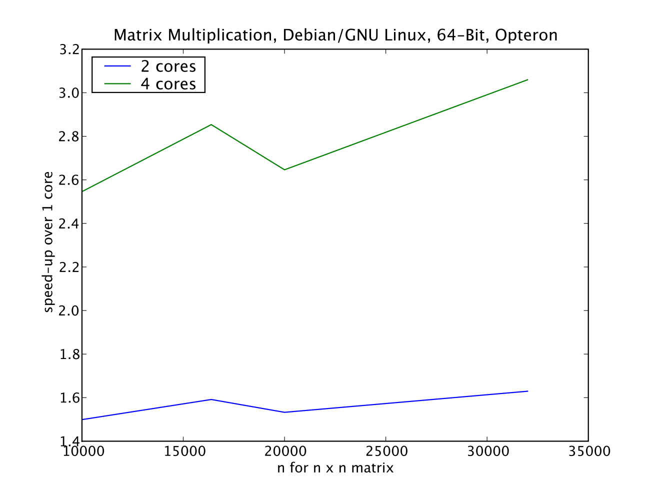


Figure: 2.6 Ghz Opteron, 18GB RAM, **L2 not shared**

Thank You!

V. Arlazarov, E. Dinic, M. Kronrod, and I. Faradzev.
On economical construction of the transitive closure of a directed graph.
Dokl. Akad. Nauk., 194(11), 1970.
(in Russian), English Translation in Soviet Math Dokl.

Gregory V. Bard.
Accelerating Cryptanalysis with the Method of Four Russians.
Cryptology ePrint Archive, Report 2006/251, 2006.
Available at `http://eprint.iacr.org/2006/251.pdf`.

L.N. Bhuyan.
CS 161 Ch 7: Memory Hierarchy Lecture 22, 1999.
Available at `http://www.cs.ucr.edu/~bhuyan/cs161/LECTURE22.ppt`.

Volker Strassen.
Gaussian elimination is not optimal.
Nummerische Mathematik, 13:354–256, 1969.