A Case Study on High Performance Matrix Multiplication

Andrés More - amore@hal.famaf.unc.edu.ar

Abstract

This document reviews a short case study on high performance matrix multiplication. Several algorithms were implemented and contrasted against a commonly used matrix multiplication library. Performance metrics including timings and operations were gathered allowing interesting graphical comparisons of the different approaches. The results showed that carefully optimized implementations are orders of magnitude above straightforward ones.

Contents

1 Introduction

This section introduces this work and the topic under study.

1.1 This Work

This work was required to complete *Cluster Programming*, a course offered as part of the *Specialization on High Performance Computing and Grid Technology* offered at *Universidad Nacional de La Plata* (UNLP)[1].

During the generation of this short report, a tool was engineered from scratch to showcase several algorithms and to extract performance metrics.

Besides the course class-notes[2], other relevant material was consulted. This included Fernando Tinetti's work [1], a discussion on architecture-aware programming [2] and also the start point of performance evaluation on clustered systems [3].

1.2 The Subject

Matrix multiplication routines are widely used in the computational sciences in general, mostly for solving linear algebra equations. Therefore, is heavily applied on scientific modeling in particular.

Timing performance is the main roadblock preventing such models to became more complex and rich enough to really match their real-world counterparts.

More than in other areas, architecture-optimized code is needed to by-pass these time barriers.

[1] `http://www.info.unlp.edu.ar`
[2] `http://ftinetti.googlepages.com/postgrado2008`

2 Algorithms

This section surveys several intuitive matrix multiplication algorithms. For each approach, code samples showing their implementation are included as example for the reader. Square matrices will be assumed to simplify the discussion.

2.1 Simple

The formal definition of matrix multiplication is shown on equation 1. Note that it only implies the final value of each element of the result matrix; nothing is stated about how values are actually calculated.

$$(AB)_{ij} = \sum_{k=1}^{k=n} a_{i1}b_{1j} = a_{i1}b_{1j} + \ldots + a_{in}b_{nj} \quad (1)$$

The previous definition can be translated into the following C implementation.

```
for (i = 0; i < n; i++)
  for (j = 0; j < n; j++)
    for (k = 0; k < n; k++)
      c[i * n + j] +=
        a[i * n + k] * b[k * n + j];
```

2.2 Blocked

An enhancing approach of the previous algorithm will be to apply the divide-and-conquer principle; it is expected that performing smaller matrix multiplications will optimize the usage of the memory cache hierarchy.

The implementation of the blocking approach is shown below.

```
for (i = 0; i < n; i += bs)
  for (j = 0; j < n; j += bs)
    for (k = 0; k < n; k += bs)
      block(&a[i * n + k],
              &b[k * n + j],
                &c[i * n + j], n, bs);
```

Where the definition of `block` implements each block multiplication, block size should be used as a boundary limit indicator.

```
for (i = 0; i < bs; i++)
  for (j = 0; j < bs; j++)
    for (k = 0; k < bs; k++)
      c[i * n + j] +=
        a[i * n + k] * b[k * n + j];
```

Note that the code above assumes that matrix size should be a multiple of block size.

2.3 Transposed

Another interesting approach [2] is to transpose the second matrix. In this way, the cache misses when iterating over the columns of the second matrix will be avoided.

On this case, each element is defined as shown in equation 2.

$$(AB)_{ij} = \sum_{k=1}^{k=n} a_{ik}b^T_{jk} = a_{i1}b^T_{j1} + \ldots + ... + a_{in}b^T_{jn} \quad (2)$$

The implementation on this case is very similar to our first approach; an additional step to rotate the matrix is added before the actual algorithm.

```
for (i = 0; i < n; i++)
  for (j = 0; j < n; j++)
    tmp[i * n + j] = b[j * n + i];

for (i = 0; i < n; i++)
  for (j = 0; j < n; j++)
    for (k = 0; k < n; k++)
      c[i * n + j] +=
        a[i * n + k] * tmp[j * n + k];
```

This technique will add overhead due to the need of the matrix rotation; it will also temporarily use more memory. Despite these drawbacks, performance gains are expected.

2.4 BLAS Library

The Basic Linear Algebra Subprograms[3] (BLAS) routines provide standard building blocks for performing basic vector and matrix operations [4]. Among its multiple routines, Level 3 BLAS routines perform general matrix-matrix operations as shown on equation 3. Where A, B, C are matrices and α, β vectors.

[3] `http://www.netlib.org/blas`

$$C \leftarrow \alpha AB + \beta C \tag{3}$$

The Automatically Tuned Linear Algebra Software (ATLAS) is a state-of-the-art, open source implementation of BLAS[4]. The library support single and multi-thread execution modes, feature that make advantage of parallelism on systems with multiple processing units.

The `dgemm` BLAS routine performs general matrix-matrix multiplication on double floating-point arithmetic, and the `cblas_dgemm` function call is a C language wrapper around the Fortran implementation. The actual library call is shown below.

```
cblas_dgemm(CblasRowMajor,
              CblasNoTrans,
                CblasNoTrans,
            n, n, n,
            1.0, a, n,
              b, n,
                0.0, c, n);
```

The first three arguments define how the arrays are stored on memory and if any matrix operand needs to be transposed. At last, the vectors and matrices to be used are provided, together with their size.

3 Results

Multiple test cases were launched to exercise the algorithms, a summary of its analysis is presented on this section.

Results were gathered on a system featuring an Intel Core Duo T2400 @ 1.83GHz CPU and also 2GB of DDR2 RAM memory; the software layer included a *Gentoo* system using the 2.6.24 *Linux* kernel and the *GCC* C compiler version 4.1.2.

The output of sample executions for each approach were logged to analyze scaling performance and timing. To calculate the work done, the obtained floating point operations per seconds (FLOPs) were computed. In addition, to get the run-time in seconds the `gettimeofday` function was used as shown below.

[4] `http://netlib.org/atlas`

```
double time;
struct timeval tv;

gettimeofday(&tv,NULL);
time = tv.tv_sec + tv.tv_usec/1000000.0;
```

3.1 Block Size

Choosing the best block size is critical on the blocked approach, this value depends on the presence and quantity of available cache memory.

Figure 1 shows experimental results while changing the block size. Best value matched the number of elements that fit inside each cache line.

To avoid binary search of the best parameter, cache line size may be used to dynamically adjust that setting.

The `sysconf(2)` POSIX interface allows to know this value at run-time; using the following code snippet is enough to find out the best block size on each system without recompilation or passing extra parameters.

```
size = sizeof(double);
cls = sysconf(_SC_LEVEL1_DCACHE_LINESIZE);
bs = cls / size;
```

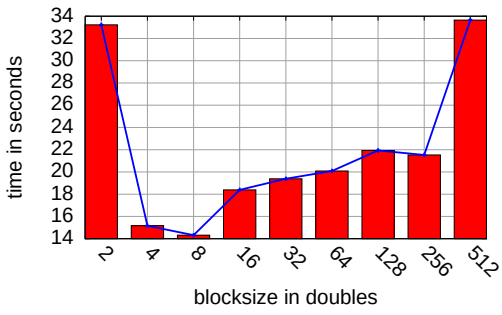

Figure 1: Block Size

3.2 Timings

According to the matrix multiplication definition, complexity will be $O(n^3)$. A better algorithm exists [5]; although it is too complex and only useful on very large sizes, not even suitable for current hardware architectures.

To exercise the implementation, all algorithms were executed for the following matrix sizes: $1024, 2048, 3072, 4096, 5120, 6144, 7168, 8192$.

The figures 3, 4, 5 and 6 show the timings of the simple, block, transp and BLAS algorithms, respectively.

Taking the 8192×8192 matrix case as example, the performance gains based on the simple definition implementation is shown on table 3.2.

method	**seconds**	**gain**
simple	25000	$\times$ 1
blocked	8000	$\times$ 3
transposed	2000	$\times$ 12
blas	500	$\times$ 50

Figure 2: 8192×8192 timings

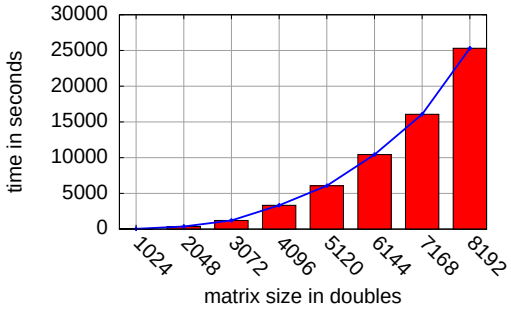

Figure 3: Simple method timings

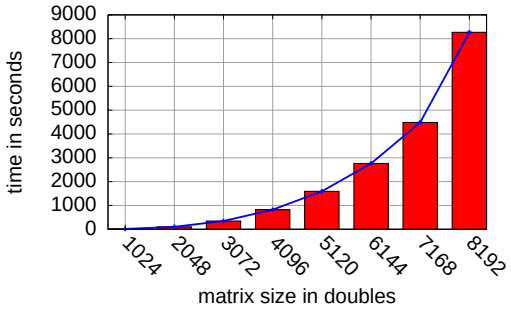

Figure 4: Block method timings

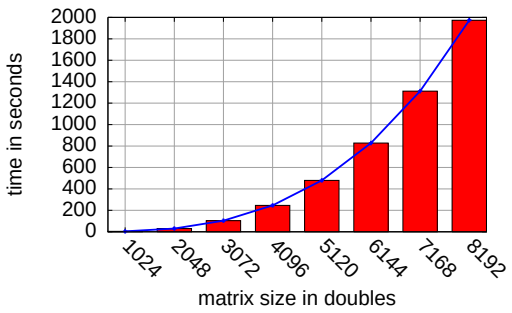

Figure 5: Transposed method timings

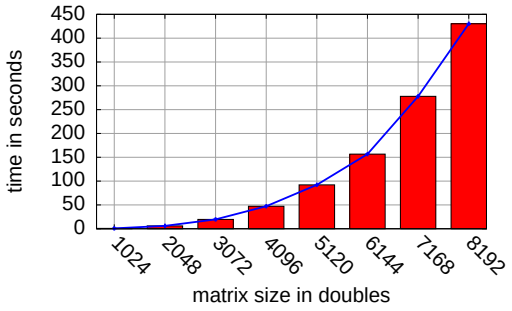

Figure 6: BLAS method timings

All results are summarized on figure 7.

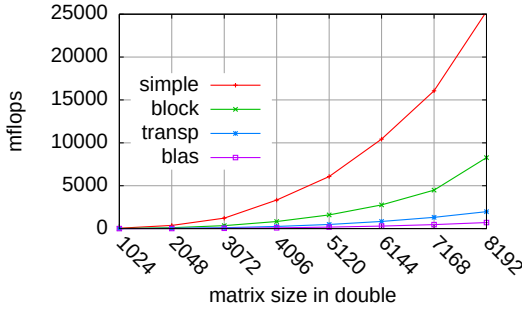

Figure 7: Timings

The exponential grow pattern is clearly depicted on each figure; imposed by the algorithms complexity. However, worst versus best implementations have differences of over 50 times.

3.3 Performance

Besides time, another interesting metric is the actual work performed per time unit.

Matrix multiplication operations can be easily estimated using formula 4, were $n \times n$ is the size of the matrix and t the time used to process it.

$$flops(method) = (n^3 - 2n^2)/t \quad (4)$$

The table below highlights overall average gains.

method	mflops	gain
simple	50	× 1
blocked	150	× 3
transposed	500	× 10
blas	1500	× 30

Figure 8: Average mflops

Figure 3.3 compares the performance of the implemented approaches. As expected all approaches showed an steady performance, and faster methods reached higher work ratio.

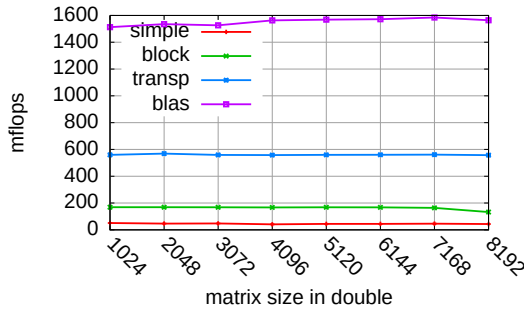

Figure 9: FLOPs Comparison

The results confirm that the system bottleneck is the memory access pattern, not the performance of the available floating point units.

4 Parallelism

Previous approaches used serial algorithms, this section details parallel solutions to the matrix multiplication problem.

4.1 Multi-threading

The ATLAS library can take advantage of multiple processing units, the multiplication is automatically spread among all available cores in the system.

Figure 10 shows a comparison between single and multi threaded BLAS executions.

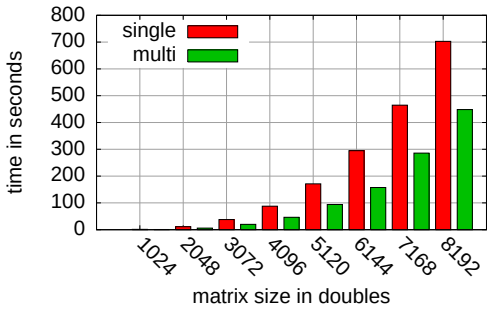

Figure 10: Single vs Multi threaded BLAS

These results shows that the ATLAS implementation has good scaling properties. Taking in account that the system has two processing units. the ideal gain is almost achieved.

4.2 Message Passing Interface

The Message Passing Interface [5] (MPI) standard defines point-to-point and collective communication among a set of processes distributed on a parallel system.

Similar to `socket(7)` programming using the `fork(2)` system call, the same binary is executed multiple times, and the programmer is in charge of the data communication protocol to be followed.

On our case, besides the actual computing of the matrix product, the code must handle the data distribution and gathering of results.

[5] http://www.mpi-forum.org

The data distribution is as follows: each worker process knows which A rows to receive (and compute) given its own task id, after computation results are sent back to the master which is in charge of the accumulation of the results into the result matrix.

The master process implementation is summarized below.

```
for (i = 1; i < size; i++) {
  MPI_Send(&a[offset * n + 0],rows * n,
             MPI_DOUBLE, i, 1, 0);
  MPI_Send(&b[0], n * n,
             MPI_DOUBLE, i, 1, 0);
  offset += rows;
}

for (i = 1; i < size; i++) {
  offset = rows * (i - 1);
  MPI_Recv(&c[offset * n + 0], rows * n,
             MPI_DOUBLE, i, 2, 0, &status)
}
```

The worker process implementation is summarized below.

```
MPI_Recv (&a[0], rows * n,
          MPI_DOUBLE, 0, 1, 0, &status);
MPI_Recv (&b[0], n * n,
          MPI_DOUBLE, 0, 1, 0, &status);

for (k = 0; k < n; k++)
  for (i = 0; i < rows; i++)
    for (j = 0; j < n; j++)
      c[i * n + k] += a[i * n + j]
                    * b[j * n + k];

MPI_Send (&c[0], rows * n,
            MPI_DOUBLE, 0, 2, 0);
```

Although more intended to other types of systems [3], a working implementation using MPI and running on the system under test was achieved. Figure 11 contains the timings gathered on such system.

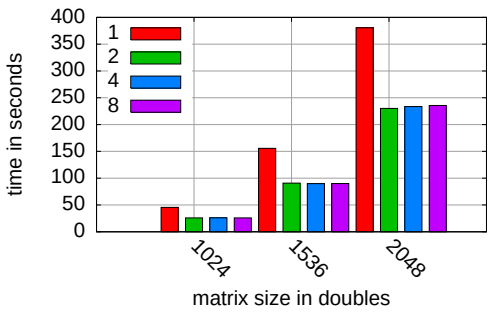

Figure 11: MPI timings according to quantity

The figure shows that using two processes nearly take half of the time. As expected, performance gain is zero when process number is greater than the quantity of processing units.

5 Conclusions

After this work, the following quick conclusions are stated.

- High performance libraries are really well optimized, their outstanding performance against mere mortals' implementations are impressive.
- High performance applications and libraries must be cache-aware, performance is highly dependant on this. To avoid recompilation, run-time information may be used for dynamic optimization.
- Theoretical approaches needs to be reviewed before the actual implementation, as the underlying architecture can alter common assumptions.
- Parallel programming is only hard, not impossible. The implementation of a distributed matrix multiplication algorithm only required the use of basic MPI communication directives and some troubleshooting.
- Memory access delays are the current bottleneck, finding a way to linearly access memory will allow easier prefetching of values and a considerable speed up.

A The mm tool

A command line tool was implemented from scratch to gather the timings and performance metrics of the different methods of matrix multiplication[6]. Serial and parallel programming approaches were included in order to analyze scaling and speed up.

The tool was made available under the GPLv2 license. Hopefully, its reuse and source code review will help to understand common issues on high performance computing.

A.1 Source Code

The source code is divided into files as shown in the table below.

file	contents
src/main.c	argument processing
src/mm.c	matrix multiplication
src/main.c	argument processing
src/utils.h	reusable definitions
src/mm.h	interface to algorithms
doc/mm.tex	document source
doc/mm.gp	gnuplot script

Figure 12: source file contents

Also, the autotools set (`autoconf`, `automake` and `libtool`) was applied for enabling a more portable build system package. The usual `configure; make; make install` is enough for deploying the tool on any supported system.

A.2 Usage

The tool support several methods for matrix multiplication, square matrices are randomly initialized and then multiplied using the requested method.

The arguments accepted include matrix size and the method to apply, optionally a check using best known implementation can be used, and also an specific block size can be requested.

Each execution output reports used time and performance achieved. The results are formatted for being processed as comma-separated value (CSV) registers. The included information follows the following syntax: `method,size,time,mflops,block,processes`.

The `argp` interface, part of the GNU C routines for argument parsing, was used for parameter handling. The tool understands short and long options, for a full list of supported options and parameters the `--usage` switch is available.

```
$ mm --usage
Usage: mm [-cv?V] [-b BLOCK] [--block=BLOCK]
          [--check] [--verbose] [--help]
          [--usage] [--version] SIZE METHOD
```

For instance, launching an execution of the BLAS implementation for two 8192×8192 matrices only requires the following command line. The `--check` option can be used to double check results during the troubleshooting of new algorithms.

```
$ mm 8192 cblas --check
cblas,8192,702.911549,1564.129257,8,1
```

References

[1] F. Tinetti. *Parallel Computing in Local Area Networks*. PhD thesis, Universidad Autonoma de Barcelona, 2004.

[2] Ulrich Deeper. What every programmer should know about memory. Technical report, Red Hat, Inc., November 2007.

[3] T. Sterling, D. Savarese, D. J. Becker, J. E. Dorband, U. A. Ranawake, and C. V. Packer. Beowulf: A parallel workstation for scientific computation. In *Proceedings of the 24th International Conference on Parallel Processing*, 1995.

[4] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic Linear Algebra Subprograms for Fortran usage. *ACM Transactions on Mathematical Software*, September 1979.

[5] D. Coppersmith and S. Winograd. Matrix multiplication via arithmetic progressions. *Journal of Symbolic Computation*, 1990.

[6] `http://code.google.com/p/mm-matrixmultiplicationtool`