DGEMM Benchmark in Manticore,  
a Heterogenous Parallel Functional Language

Liem Radita Tapaning Hesti  
Institute of Computer Science University of Tartu  
Tartu, Estonia  
Email: hesti@ut.ee

Abstract—DGEMM benchmark is created using Manticore, a heterogeneous functional programming language that supports parallelism at multiple levels. Manticore is aimed to be a high-level parallel programming language targeted for typical commodity microprocessor. The benchmarking result is compared with the existing DGEMM library running in the same Intel Core i5 machine.

1. Introduction

The integral part of high-performance computing (HPC) is measuring and reporting performance of parallel computing systems. The measurement can cover multiple aspects, from hardware performance to programming language performance. In this work, Manticore parallel programming language performance will be measured using Double-precision General Matrix Multiplication (DGEMM) benchmarking.

Dense matrix operations are an important element in scientific and engineering computing applications and due to its importance, there are a lot of high-performance libraries for dense matrix calculations. One of the popular libraries for dense matrix multiplication is Basic Linear Algebra Subprograms (BLAS). BLAS is a standard application programming interface to execute basic linear algebra operations such as vector and matrix multiplication [4]. In this project, BLAS provided Intel Math Kernel Library (MKL) DGEMM library will be used and the result collected will be compared with the result from Manticore DGEMM implementation.

1.1. Manticore Programming Language

Manticore is a heterogeneous functional programming language that supports parallelism at multiple levels. It supports both explicit concurrency and implicit parallelism. At the explicit-concurrency level, distinct threads of control can be done and the coordination happens through first-class synchronous-message passing. Message-passing synchronization, in contrast to shared-memory synchronization, is more suitable for the functional programming paradigm. At the implicit-parallelism level, a diverse collection of parallel constructs is supported for different granularities of work.

The core language is based on the Standard ML (SML) language. The main differences are that Manticore does not have mutable data (i.e., reference cells and arrays) and, in the present language implementation, Manticore has a simplified module system (omitting functors and sophisticated type sharing). Manticore has the functional elements of SML such as datatypes, polymorphism, type inference, and higher-order functions, as well as exceptions [1].

1.2. Intel Math Kernel Library (Intel MKL)

Hardware vendors usually have provided BLAS libraries tuned on their own architecture. In Intel architecture, there is Intel Math Kernel Library (Intel MKL). The Intel MKL contains many routines to solve various numerical problems, such as matrix multiplication, a system of equations, and Fourier transform. Not all problems have its dedicated routines and for these problems, it can be solved by assembling the building blocks provided by Intel MKL.

1.3. Double Precision Matrix Multiplication

Matrix multiplication defines C = A*B, where A, B, and C are m x k, k x n, m x n matrices, respectively. A naive implementation of matrix multiplications is three nested loops that produce n^3 multiplications operations. The basic matrix multiplication algorithm is as follow:

Listing 1. Naive Matrix Multiplication Algorithm

```c
for (int i = 0; i < BLOCK_SIZE ; i++) 1
    for (int j = 0; j < BLOCK_SIZE ; j++) 2
        for (int k = 0; k < BLOCK_SIZE ; k++) 3
            C[i][j]+=A[i][k]+B[k][j];
```

In this work, all matrices is a square matrices with all the elements of the matrices are randomly generated as double precisions numbers. Naive implementation mentioned above is the base of the code in this experiment.

2. Implementation procedure

This work will use six sizes of matrices (1024 x 1024, 2048 x 2048, 4096 x 4096, 8192 x 8192, 16384 x 16384,
2.1. Sequential Implementation

To ensure if the parallelism works correctly, a benchmark with sequential matrix multiplication is created. The matrix multiplication sequential program is in Standard Meta Language (Standard ML) so it can be compared to Manticore parallel implementation.

Function headcol gets the first value of each row in the matrix and return it as a list. tailcols returns the same matrix without the first value of its each row. transposition and dot product are respectively done by transp and dotprod. The final result is assembled in matprod.

Below is the code taken from Standard ML Tutorial Nodes from Technion institute of Technology, Israel [5].

Listing 2. Sequential matrix multiplication

```ml
fun headcol [] = []
| headcol ((x::[]):rows) = x :: headcol rows;

fun tailcols [] = []
| tailcols ((_:xs):rows) = xs::tailcols rows;

fun dotprod([],[],0.0) = 0.0
| dotprod(xs,ys) = x*y+dotprod(xs,ys);

fun transp ([]:rows) = []
| transp rows = headcol rows ::
| transp (tailcols rows);

fun rowprod(row,[]) = []
| rowprod(row,cols)=
| dotprod(row,cols)::rowprod(row,cols);

fun rowlistprod([],cols) = []
| rowlistprod(row::rows,cols)=
| rowlistprod(row::rows,cols)::rowlistprod(rows,cols);

fun matprod (Arows,Brows) =
| rowlistprod(Arows,transp Brows);
```

2.2. Manticore Parallel Matrix Multiplication

In the parallel implementation, implicit parallelism using PArray.reduce performs a tree-shaped collapsing reduction over a rope of double-precision floating-point numbers in Sum function. The sums of the values at each leaf are computed sequentially, and the sums of the values under each internal node are added to one another once they have both been computed. This process happens over the whole tree in parallel, and the sum of the whole is the sum computed at the rope’s topmost node.

The function dotp is the application of sum to multiplication operations, with a parallel array of double representing a dense vector, over a parallel array of pairs. The function smvm is an application of dotp over the members of an array of such arrays of pairs. Below code is taken from Sparse Matrix Multiplication on Manticore [6]:

```
Listing 3. Matrix multiplication in Manticore

fun plus (x:double, y:double) = x+y
fun sum (xs : double parray) = PArray.reduce plus (0.0) xs
fun dotp (sv, v) = sum [[ x * (v!i) | (i,x) in sv ]] [i, x in sv ]
fun smvm (sm, v) = [ | dotp (sv, v) | sv in sm ]
```

2.3. Intel MKL DGEMM in C

Intel MKL has its own DGEMM library that can be called using cblas_dgemm function. Before the execution, it requires memory allocation along with matrix size and scalar precision value. Code below is the example provided by Intel Developer Zone [7]:

```
Listing 4. Matrix multiplication in Intel MKL

A = (double *)malloc( m*k*sizeof( double ), 64 );
B = (double *)malloc( k*n*sizeof( double ), 64 );
C = (double *)malloc( m*n*sizeof( double ), 64 );

for (i = 0; i < (m*k); i++) {
    A[i] = (double)(i+1);
}
for (i = 0; i < (k*n); i++) {
    B[i] = (double)(-i-1);
}
for (i = 0; i < (m*n); i++) {
    C[i] = 0.0;
}
cblas_dgemm(BlasNoTrans, BlasNoTrans, m, n, k, alpha, A, k, B, n, beta, C, n);

mkl_free(A);
Mkl_free(B);
Mkl_free(C);
```

3. Result

The result shows that Intel MKL DGEMM library consumed the least execution time compared to Manticore parallel and sequential approach. Intel MKL library has
the best performance result is expected since the library is already tuned to the machine architecture. The sequential implementation gets the result as expected where its execution time grows exponentially following its matrix size. Parallel Manticore code shows a linear result as the matrix size grows. This result aligns with Intel MKL DGEMM result that is also linear.

4. Conclusion

Even though the parallel implementation cannot beat Intel DGEMM Library, Manticore parallelization offers ease of use and concise syntax due to its functional programming approach. Manticore parallel code works as expected to keep execution time linear as the matrix size grows.

5. Future Work

The manticore code is designed to do sparse matrix multiplication problem thus might be not working properly in dense matrix multiplication problem. The parallel program is also not yet tuned to the machine to increase its performance. These concerns can be addressed in future work.

References


