International Journal of Differential Equations and Applications
Volume **20**, No. 2 (2021), pages: 217-223
ISSN (Print): 1311-2872; ISSN (Online): 1314-6084;
url: https://www.ijdea.eu

EFFICIENT MATRIX MULTIPLICATION USING HARDWARE INTRINSICS AND PARALLELISM WITH C#

Nikolay Pavlov

Faculty of Mathematics and Informatics
Paisii Hilendarski University of Plovdiv
236 Bulgaria Blvd., 4003 Plovdiv, BULGARIA

ABSTRACT: In this paper we improve the efficiency of the simple matrix-multiplication algorithm using parallelism and hardware instrinsics with C# and .Net Task Parallel Library. We demonstrate how additional processing can result the overall shorter execution times. We present an algorithm in C# that achieves up to five times performance improvement on midrange hardware, using single-instruction-multiple-data instructions and unsafe code foo direct access to memory where other techniques are not possible in the .Net managed environment.

AMS Subject Classification: 68W10
Key Words: matrix multiplication, hardware instrinsics, parallelism, simd, multithreading, .net

Received: August 13, 2021 **Revised:** November 1, 2021
Published: November 15, 2021 **doi:** 10.12732/ijdea.v20i2.8
Academic Publications, Ltd. https://acadpubl.eu

1. INTRODUCTION

Matrix Multiplication is one of the most fundamental operation in machine learning. The naïve m atrix multiplication algorithm for square matrices looks like this (in C#):

```
public double[,] MultiplyMaticesNaive(double[,] m1, double[,] m2)
{
  int size = m1.GetUpperBound(0);
  double[,] result = new double[size, size];
  for (int i = 0; i < size; i++)
  {
    for (int j = 0; j < size; j++)
    {
      for (int k = 0; k < size; k++)
      {
        result[i, j] += m1[i, k] * m2[k, j];
      }
     }
  }
  return result;
}
```

The algorithm has $O(n3)$ for square matrices with size n.

There exist improved algorithms, most notable of which is Strassen's matrix multiplication algorithm [1], which served as a base for further research. Generally Strassen's algorithm is not preferred for practical applications for the following reasons: the submatrices in recursion take extra space and execution costs, and because of the limited precision of computer arithmetic on non-integer values, larger errors accumulate in Strassen's algorithm than in the naïve algorithm described above [2]. In 2020 Alman and Williams created an algorithm that achieves $O(n2.37287)$ complexity [3], but the algorithm is impractical for real use along with other similar [4].

In this paper we take the practical approach to improve the naïve algorithm using parallelism and knowledge about the popular hardware, i.e., intrinsics.

2. PARALLELISM

We can make use of the multiple cores in modern CPUs and create a parallel version of the algorithm. For simplicity, we use the Task Parallel Library [5, 6] of Microsoft .NET.

```
public double[,] MultiplyMaticesParallel(double[,] m1, double[,] m2)
{
  int size = m1.GetUpperBound(0);
  double[,] result = new double[size, size];
  Parallel.For(0, size, i =>
  {
```

```
    for (int j = 0; j < size; j++)
    {
      for (int k = 0; k < size; k++)
      {
        result[i, j] += m1[i, k] * m2[k, j];
      }
    }
  });
  return result;
}
```

Using parallelism, we can see an improvement of more than double efficiency: 442.6 ms versus 1128.7 ms for a matrix of size 600.

3. IMPROVED USE OF HARDWARE CACHE

From hardware point of view, this implementation has a problem with utilization of CPU cache, because data in $m2$ is not accessed sequentially. An improvement is suggested by Drepper [7] that rearranges matrix $m2$ before running the algorithm:

```
double[,] TranspositeMatrix(double[,] m)
{
  int size = m.GetUpperBound(0);
  double[,] result = new double[size, size];
  for (int i = 0; i < size; i++)
  {
    for (int j = 0; j < size; j++)
    {
      result[i, j] = m[j, i];
    }
  }
  return result;
}
```

Combining the two increases the overall complexity to $O(n^3 + n^2)$ but we achieve an improvement of about 17% : 369.7 versus 442.6 ms.

4. ALGORITHM BASED ON SINGLE-INSTRUCTION-MULTIPLE-DATA (SIMD)

We present an enhanced version of the algorithm, implemented in the prior sections. We focus on hardware intrinsics, or hardware-specific implementation, namely the spe-

cialized data parallel processing instructions – SIMD [8], present in modern CPUs. Using SIMD, we reduce the number of iterations in the most inner loop of the algorithm. While similar research has been carried out [], it has focused on C or specialized libraries for multiple data processing. Now we present a SIMD-based algorithm in C# and NET 5.

In .NET 5, SIMD instructions are available via the System. Numeric.Vector type.

For simplicity, we skip the range checks, and the algorithm assumes that the number of columns in the input matrices is divisible by four. We also assume a maximum parallelism of 256 bits; hence the step of the most inner loop is set to 4.

```
public double[,] MultiplyMaticesTransParallelVector(double[,] m1, double
    [,] m2)
{
  int size = m1.GetUpperBound(0);
  double[,] m2T = TranspositionMatrix(m2);
  double[,] result = new double[size, size];
  Parallel.For(0, size, i =>
  {
    double[] pack1 = new double[4], pack2 = new double[4];
    for (int j = 0; j < size; j++)
    {
      for (int k = 0; k < size-4; k+=4)
      {
        pack1[0] = m1[i, k]; pack2[0] = m2T[j, k];
        pack1[1] = m1[i, k + 1]; pack2[1] = m2T[j, k + 1];
        pack1[2] = m1[i, k + 2]; pack2[2] = m2T[j, k + 2];
        pack1[3] = m1[i, k + 3]; pack2[3] = m2T[j, k + 3];
        Vector<double> vMul =
          Vector.Multiply(new Vector<double>(pack1), new Vector<double>(
    pack2));
        result[i, j] += vMul[0] + vMul[1] + vMul[2] + vMul[3];
      }
    }
  });
  return result;
}
```

It is obvious that the algorithm above has an excessive number of index computations, which cannot be mitigated with a loop – the loop will in effect achieve the same number of computations but add also branching instructions. An alternative is to use direct access to memory via the new `Span<T>` type of NET, which enables the Just-in-Time (JIT) compiler to utilize SIMD instructions for copying data from the

array to the temporary buffers `pack1` and `pack2`.

A version of the algorithm working with jagged arrays, or an array-of arrays, would allow a direct use of `Span<T>`. However, our algorithm uses two-dimensional arrays for the input and output parameters which cannot be accessed via `Span<T>`. We can mitigate this issue using unsafe code:

```
public double[,] MultiplyMaticesTransParallelVectorUnsafe(double[,] m1,
    double[,] m2)
{
  int size = m1.GetUpperBound(0);
  double[,] m2T = TranspositionMatrix(m2);
  double[,] result = new double[size, size];
  Parallel.For(0, size, i =>
  {
    for (int j = 0; j < size; j++)
    {
      Span<double> s1, s2;
      for (int k = 0; k < size - 4; k += 4)
      {
        unsafe
        {
          fixed (double* p = &m1[i, k])
          {
            s1 = new Span<double>(p, 4);
          }
          fixed (double* p = &m2T[j, k])
          {
            s2 = new Span<double>(p, 4);
          }
        }
        Vector<double> vMul =
           Vector.Multiply(new Vector<double>(s1), new Vector<double>(s2
    ));
        result[i, j] += vMul[0];
        result[i, j] += vMul[1];
        result[i, j] += vMul[2];
        result[i, j] += vMul[3];
      }
    }
  });
  return result;
}
```

Unsafe code enables us to directly access the memory at the double index of the two-dimensional array. Reducing the amount of work in the most inner loop yields

further improvement of about 25%.

5. RESULTS

The following results were achieved for multiplying two square matrices containing double-precision floating point values of size 600.

Algorithm	**Time (ms)**	**Ratio**
Naïve (MultiplyMaticesNaive)	1128.7	1.00
Parallel (MultiplyMaticesParallel)	442.6	0.39
Transpose + Parallel	369.7	0.31
Transpose + Parallel + SIMD (MultiplyMaticesParallelVector)	291.4	0.26
Transpose + Parallel + SIMD + Unsafe (MultiplyMaticesTransParallelVectorUnsafe)	218.3	0.19

Table 1. Benchmark Results

Performance was measured using tool BenchmarkDotNet [10, 11], version 0.12.1 on a single Intel Core i7-8550U (Kaby Lake R, 8 logical and 4 physical cores) CPU running at 1.80 GHz (max turbo – 4.00 GHz). We use .NET version 5.0 and x64 RyuJIT JIT compiler.

6. CONCLUSION

In this paper we present a matrix multiplication algorithm in C# and NET 5 that makes use of parallelism and various techniques to make optimal use of the hardware to achieve up to five times improvements in performance over the native implementation of the matrix multiplication algorithm.

ACKNOWLEDGMENTS

This paper is supported by the National Scientific Program “Information and Communication Technologies for Unified Digital Market in Science, Education and Security (ICTinSES)”, financed by the Ministry of Education and Science.

REFERENCES

[1] V. Strassen, Gaussian elimination is not optimal, *Numerische Mathematik*, (1969), **13**, 354–356, DOI: 10.1007/BF02165411.

[2] W. Miller, Computational Complexity and Numerical Stability, *SIAM Journal on Computing*, (1975), Vol. **4** (2), 97–107, ISSN: 0097-5397, DOI: 10.1137/0204009.

[3] J. Alman, V. Williams, A Refined Laser Method and Faster Matrix Multiplication, *32nd Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, (2021), arXiv:2010.05846.

[4] F. Le Gall, Faster algorithms for rectangular matrix multiplication, *Proceedings of the 53rd Annual IEEE Symposium on Foundations of Computer Science (FOCS 2012)*, 514–523, arXiv:1204.1111, DOI:10.1109/FOCS.2012.80.

[5] J. Duffy, *Concurrent Programming on Windows*, Addison Wesley, (2009), ISBN: 978-0-321-43482-1.

[6] Microsoft, Task Parallel Library, accessed Sep 2021, https://docs.microsoft.com/en-us/dotnet/standard/parallel-programming/task-parallel-library-tpl.

[7] U. Drepper, *What every programmer should know about memory*, Red Hat, Inc., (2007).

[8] M. Flynn, Some Computer Organizations and Their Effectiveness, IEEE Transactions on Computers, (1972), Vol. **C-21**, No. 9, 948–960, DOI: 10.1109/TC.1972.5009071.

[9] K. Goto, R. Geijn, Anatomy of High-Performance Matrix Multiplication, *ACM Transactions on Mathematical Software*, May 2008, Vol. **34**, 3, Article 12, DOI 10.1145/1356052.1356053.

[10] https://benchmarkdotnet.org, last visited September 2021.

[11] A. Akinshin, *Pro .NET Benchmarking*, Apress, (2019), ISBN: 978-1-4842-4940-6, DOI: 10.1007/978-1-4842-4941-3.