ECE 6374 PARALLEL COMPUTATIONS

Assignment 1 Report

By

Bijon Sahu(1647781)

&

Manoj Kumar Cebol Sundarrajan(1620546)

Date

02/27/2018

**MATRIX-MATRIX MULTIPLICATION OPTIMIZATION**

An optimized single threaded matrix multiplication kernel on the crill cluster for the AMD Opteron 6174 processor which has a peak performance 2.2GHz(3.2GHz) \* 2 elements/vector \* 2 operations per FMA=8.8GFLOP/s (12.8GFLOP/s) (number in parentheses are with turbo frequency boost). The loop which we will be optimizing is

void square\_dgemm (int n, double\* A, double\* B, double\* C)

{

/\* For each row i of A \*/

for ( int i = 0; i < n; ++i)

{

/\* For each column j of B \*/

for (int j = 0; j < n; ++j)

{

/\* Compute C(i,j) \*/

double cij = C[i+j\*n];

for( int k = 0; k < n; k++ )

{

cij += A[i+k\*n] \* B[k+j\*n];

C[i+j\*n] = cij;

}

}

}

The processor characteristics are as follows

Architecture: x86\_64

CPU op-mode(s): 32-bit, 64-bit

Byte Order: Little Endian

CPU(s): 8

On-line CPU(s) list: 0-7

Thread(s) per core: 1

Core(s) per socket: 4

Socket(s): 2

NUMA node(s): 2

Vendor ID: AuthenticAMD

CPU family: 16

Model: 2

Model name: Quad-Core AMD Opteron(tm) Processor 2354

Stepping: 3

CPU MHz: 2200.000

CPU max MHz: 2200.0000

CPU min MHz: 1100.0000

BogoMIPS: 4423.12

Virtualization: AMD-V

L1d cache: 64K

L1i cache: 64K

L2 cache: 512K

L3 cache: 2048K

NUMA node0 CPU(s): 0-3

NUMA node1 CPU(s): 4-7

Flags: fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush mmx fxsr sse sse2 ht syscall nx mmxext fxsr\_opt pdpe1gb rdtscp lm 3dnowext 3dnow constant\_tsc rep\_good nopl nonstop\_tsc extd\_apicid pni monitor cx16 popcnt lahf\_lm cmp\_legacy svm extapic cr8\_legacy abm sse4a misalignsse 3dnowprefetch osvw ibs hw\_pstate npt lbrv svm\_lock vmmcall

The basic naïve code was optimized through various stages as shown below

**1. NAÏVE Code(without any optimization):**

The naive unblocked routine is just the standard three nested loops.

int sum=0;

for(int i=0;i<n;i++)

{

for(int j=0;j<n;j++)

{

sum = c[i][j];

for(int k=0;k<n;k++)

{

sum += a[i][k] \* b[k][j];

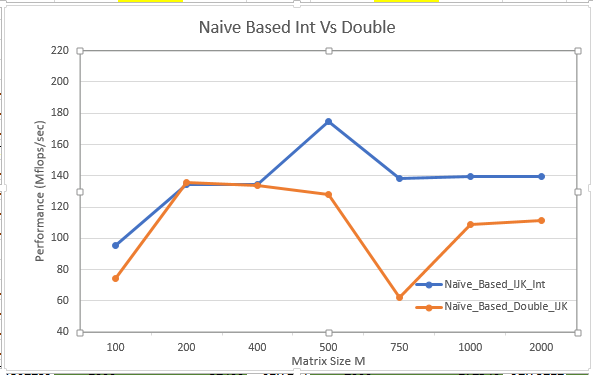
}

c[i][j] = sum;

}

}

The performance of naïve code is really low as compared to the machine peak performance. Later in this article, other optimizations would be compared as performance increase to naïve implementation.



**Fig 1 : Shows NAVIE based performance for different Matrix sizes of both INT and DOUBLE type**

**2. L1 Cache Blocking :**

L1 Cache of the AMD processor consists of 64KB of memory. The blocking size can be determined using the formula

*Where B is the block size.*

Solving,

It is found that Block Size for int(2 byte) is 73 elements and for double(8 byte) is 51 elements.

The implemented for loop is

For (int i1=0; i1<n; i1+=BLOCK\_SIZE)

{

      For (int j1=0; j1<n; j1+=BLOCK\_SIZE)

      {

For (int k1=0; k1<n; k1+=BLOCK\_SIZE)

          {

              For (int i=i1; i<i1+BLOCK\_SIZE; i++)

              {

                  For ( int j=j1; j<j1+BLOCK\_SIZE; j++)

                  {

                      sum = c[i][j];

                      For ( int k=k1; k<k1+BLOCK\_SIZE; k++)

                      {

                          sum += a[i][k] \* b[k][j];

                      }

                      c[i][j] = sum;

                  }

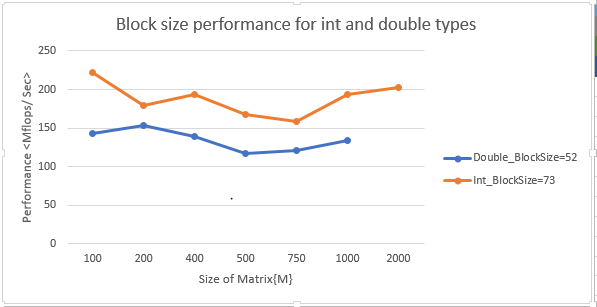
}

          }

      }

}

The performance can be visualized in the graph shown.



**Fig 2 : Shows L1 cache blocking based performance for different Matrix sizes of both**

**INT and DOUBLE type**

**3. Optimization using Transpose of B Matrix:**

In naïve Matrix multiplication, Matrix B is accessed in column order. Which accounts for lots of cache misses. So the idea is to get the transpose of the matrix so the elements will be accessed in row order.

The loop is as follows

for(int i=0;i<n;i++)

{

  for(int j=0;j<n;j++)

              {

                      sum = c[i][j];

                      for(int k=0;k<n;k++)

                      {

                          sum += a[i][k] \* b[k][j];

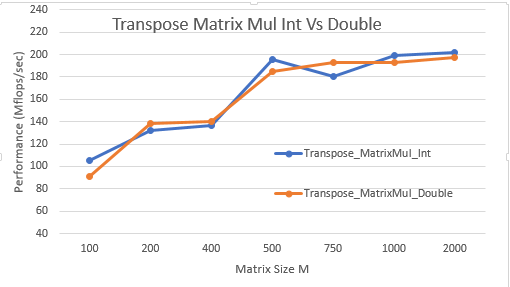
                      }

                      c[i][j] = sum;

                  }

}

The performance in FLOPS can be graphed as follows



**Fig 3 : Shows Transpose Matrix Multiplication based performance for different Matrix**

**Sizes of both**

**INT and DOUBLE type**

**4. Optimization using Loop interchange:**

In this optimization we will be toggling between i, j, k loops. In naïve code, the loops are order as I, j, k. Let us look on other order of the loops.

**4.1. jik optimization:**

Here, the for loops are placed as follows

/\* jik \*/

for (j=0; j<n; j++)

{

for (i=0; i<n; i++)

{

sum = 0.0;

for (k=0; k<n; k++)

{

sum += a[i][k] \* b[k][j];

c[i][j] = sum

}

}

}

**4.2. kij optimization:**

Kij optimized loop is

/\* kij \*/

for (k=0; k<n; k++)

{

for (i=0; i<n; i++)

{

r = a[i][k];

for (j=0; j<n; j++)

{

c[i][j] += r \* b[k][j];

}

}

}

**4.3. ikj optimization:**

Ikj optimized loop

/\* ikj \*/

for (i=0; i<n; i++)

{

for (k=0; k<n; k++)

{

r = a[i][k];

for (j=0; j<n; j++)

{

c[i][j] += r \* b[k][j];

}

}

}

**4.4. jki optimization:**

/\* jki \*/

for (j=0; j<n; j++)

{

for (k=0; k<n; k++)

{

r = b[k][j];

for (i=0; i<n; i++)

{

c[i][j] += a[i][k] \* r;

}

}

}

**4.5. kji optimization:**

/\* kji \*/

for (k=0; k<n; k++)

{

for (j=0; j<n; j++)

{

r = b[k][j];

for (i=0; i<n; i++)

{

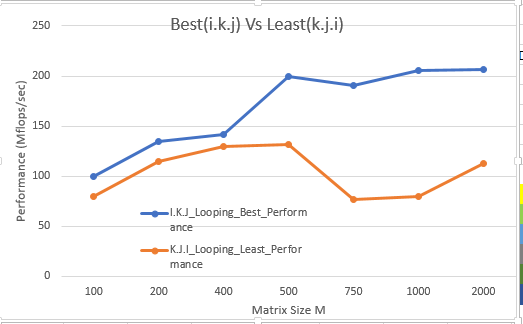
c[i][j] += a[i][k] \* r;

}

}

}

Out of all possibilities of I, J, K looping probability we could see that performance of best loop (i.k.j) Vs least performative loop (k.j.i)



**Fig 4 : Shows Looping Interchange multiplication matrix based on best and least performances.**

Some more optimizations, which may lead to marginal improvements, are as follows:

**1. Software pre-fetch:** A lot of cache misses are intrinsic misses, during which the L1, L2 caches are being loaded with data from the memory. Pre- fetch can help reduce these latencies. Usually compilers do not support these techniques and hence these operations have to be done in assembly level code, which makes it difficult to implement them. Below mentioned is a code in C which uses <smmintrin.h> header and \_m128 object type to instantiate.

#include <stdio.h>

#include <smmintrin.h>

int main ()

{

\_\_m128 a, b;

const int mask = 0x55;

a.m128\_f32[0] = 1.5;

a.m128\_f32[1] = 10.25;

a.m128\_f32[2] = -11.0625;

a.m128\_f32[3] = 81.0;

b.m128\_f32[0] = -1.5;

b.m128\_f32[1] = 3.125;

b.m128\_f32[2] = -50.5;

b.m128\_f32[3] = 100.0;

\_\_m128 res = \_mm\_dp\_ps(a, b, mask);

printf\_s("Original a: %f\t%f\t%f\t%f\nOriginal b: %f\t%f\t%f\t%f\n",

a.m128\_f32[0], a.m128\_f32[1], a.m128\_f32[2], a.m128\_f32[3],

b.m128\_f32[0], b.m128\_f32[1], b.m128\_f32[2], b.m128\_f32[3]);

printf\_s("Result res: %f\t%f\t%f\t%f\n",

res.m128\_f32[0], res.m128\_f32[1], res.m128\_f32[2], res.m128\_f32[3]);

return 0;

}

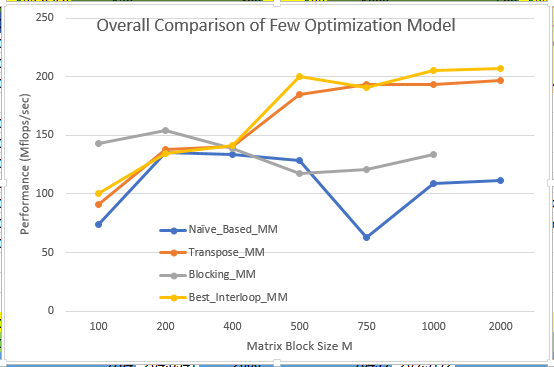
**2. Loop unrolling and register reuse:** Most of the loops were then unrolled to speed the performance. Also registers were used explicitly to store some quite frequent variables so that they do not have to be repeatedly fetched from the L1 or L2 caches.

**3. Vectorization:** This property would allow us to pack several arithmetic operations into a single vector instruction. While most of our previous optimizations have focused on reducing

total memory access latency, these instructions would allow us to speed up computation time.

**Conclusion:**

As we could only perform testing on few of the optimization models , we could only compare and limited set of result as shown below which indicates that blocking has better performance until the matrix size 400 but then transpose has much better for size greater than 400 to 2K.



**Fig 5 : Show overall results of few optimization programs**

**Instructions to run on crill:**

1. Compile the code using the following command line

g++ -o output [filename].c

2. Create a batch script as follows

#!/bin/bash

#SBATCH -J Prog1

#SBATCH -n 1

#SBATCH -N 1

#SBATCH -p cosc6374

./output

3. Run the batch script using SBATCH command

sbatch ./[batchscriptname]

4. View the output in the slurm-[jobid].out file