CS639 Assignment 2 Report

Computer used = CSLab Machine vm-instunix-01

Architecture: x86\_64

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

CPU(s): 12

On-line CPU(s) list: 0-11

Vendor ID: GenuineIntel

Model name: 11th Gen Intel(R) Core(TM) i5-11500 @ 2.70GHz

CPU family: 6

Model: 167

Thread(s) per core: 2

Core(s) per socket: 6

Socket(s): 1

Stepping: 1

CPU max MHz: 4600.0000

CPU min MHz: 800.0000

BogoMIPS: 5424.00

How I compiled the code = icc main.cpp Laplacian.cpp ConjugateGradients.cpp PointwiseOps.cpp Reductions.cpp Utilities.cpp MatVecMultiply.cpp -O3 -qopenmp -mkl -o main.exe

What kernels I modified =

The ***InnerProduct()*** call.

The ***Copy()*** call.

The ***Norm()*** call.

Files modified =

PointwiseOps.cpp

#include "PointwiseOps.h"

// #define DO\_NOT\_USE\_MKL

#ifndef DO\_NOT\_USE\_MKL

#include <mkl.h>

#endif

void Copy(const float (&x)[XDIM][YDIM][ZDIM], float (&y)[XDIM][YDIM][ZDIM])

{

#ifdef DO\_NOT\_USE\_MKL

#pragma omp parallel for

for (int i = 1; i < XDIM-1; i++)

for (int j = 1; j < YDIM-1; j++)

for (int k = 1; k < ZDIM-1; k++)

y[i][j][k] = x[i][j][k];

#else

int size = XDIM \* YDIM \* ZDIM;

cblas\_scopy(

size,

&x[0][0][0],

1,

&y[0][0][0],

1

);

#endif

}

void Saxpy(const float (&x)[XDIM][YDIM][ZDIM], const float (&y)[XDIM][YDIM][ZDIM],

float (&z)[XDIM][YDIM][ZDIM],

const float scale)

{

#pragma omp parallel for

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

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

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

z[i][j][k] = x[i][j][k] \* scale + y[i][j][k];

}

void Saxpy(const float (&x)[XDIM][YDIM][ZDIM], float (&y)[XDIM][YDIM][ZDIM],

const float scale)

{

#ifdef DO\_NOT\_USE\_MKL

// Just for reference -- implementation without MKL

#pragma omp parallel for

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

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

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

y[i][j][k] += x[i][j][k] \* scale;

#else

cblas\_saxpy(

XDIM \* YDIM \* ZDIM, // Length of vectors

scale, // Scale factor

&x[0][0][0], // Input vector x, in operation y := x \* scale + y

1, // Use step 1 for x

&y[0][0][0], // Input/output vector y, in operation y := x \* scale + y

1 // Use step 2 for y

);

#endif

}

Reductions.cpp

#include "Reductions.h"

#include <algorithm>

#include <iostream>

// #define DO\_NOT\_USE\_MKL

#ifndef DO\_NOT\_USE\_MKL

#include <mkl.h>

#endif

float Norm(const float (&x)[XDIM][YDIM][ZDIM])

{

#ifdef DO\_NOT\_USE\_MKL

float result = 0.;

#pragma omp parallel for reduction(max:result)

for (int i = 1; i < XDIM-1; i++)

for (int j = 1; j < YDIM-1; j++)

for (int k = 1; k < ZDIM-1; k++)

result = std::max(result, std::abs(x[i][j][k]));

return result;

#else

int index =

cblas\_isamax(

XDIM \* YDIM \* ZDIM,

&x[0][0][0],

1

);

int i = index / (YDIM \* ZDIM);

int j = (index / ZDIM) % YDIM;

int k = index % ZDIM;

return std::abs(x[i][j][k]);

#endif

}

float InnerProduct(const float (&x)[XDIM][YDIM][ZDIM], const float (&y)[XDIM][YDIM][ZDIM])

{

double result = 0.;

#ifdef DO\_NOT\_USE\_MKL

#pragma omp parallel for reduction(+:result)

for (int i = 1; i < XDIM-1; i++)

for (int j = 1; j < YDIM-1; j++)

for (int k = 1; k < ZDIM-1; k++)

result += (double) x[i][j][k] \* (double) y[i][j][k];

return (float) result;

#else

int size = XDIM \* YDIM \* ZDIM;

return cblas\_sdot(

size,

&x[0][0][0],

1,

&y[0][0][0],

1

);

#endif

}

main.cpp

(Added a timerTotal to time the total runtime of ConjugateGradients)

#include "ConjugateGradients.h"

#include "Laplacian.h"

#include "Timer.h"

#include "Utilities.h"

Timer timerLaplacian;

Timer timerSaxpy;

int main(int argc, char \*argv[])

{

using array\_t = float (&) [XDIM][YDIM][ZDIM];

float \*xRaw = new float [XDIM\*YDIM\*ZDIM];

float \*fRaw = new float [XDIM\*YDIM\*ZDIM];

float \*pRaw = new float [XDIM\*YDIM\*ZDIM];

float \*rRaw = new float [XDIM\*YDIM\*ZDIM];

float \*zRaw = new float [XDIM\*YDIM\*ZDIM];

array\_t x = reinterpret\_cast<array\_t>(\*xRaw);

array\_t f = reinterpret\_cast<array\_t>(\*fRaw);

array\_t p = reinterpret\_cast<array\_t>(\*pRaw);

array\_t r = reinterpret\_cast<array\_t>(\*rRaw);

array\_t z = reinterpret\_cast<array\_t>(\*zRaw);

CSRMatrix matrix;

// Initialization

{

Timer timer;

timer.Start();

InitializeProblem(x, f);

matrix = BuildLaplacianMatrix(); // This takes a while ...

timer.Stop("Initialization : ");

}

Timer timerTotal;

timerTotal.Start();

// Call Conjugate Gradients algorithm

{

timerLaplacian.Reset(); timerSaxpy.Reset();

ConjugateGradients(matrix, x, f, p, r, z, false);

timerLaplacian.Print("Total Laplacian Time : ");

timerSaxpy.Print("Total Saxpy Time : ");

}

timerTotal.Stop("Total time: ");

return 0;

}

Without Using MKL Kernels, runtime and residual norm was:

Runtime = [Total time: 36194.8ms]

Residual Norm = Conjugate Gradients terminated after 256 iterations; residual norm (nu) = 0.000975894

Using MKL Kernels:

Runtime = [Total time: 27137.5ms]

Residual Norm = Conjugate Gradients terminated after 256 iterations; residual norm (nu) = 0.000974475

I think this is saying that the MKL Kernels are performing better than the original ones. Also it’s correct because the number of iterations and final residual norm are very close.