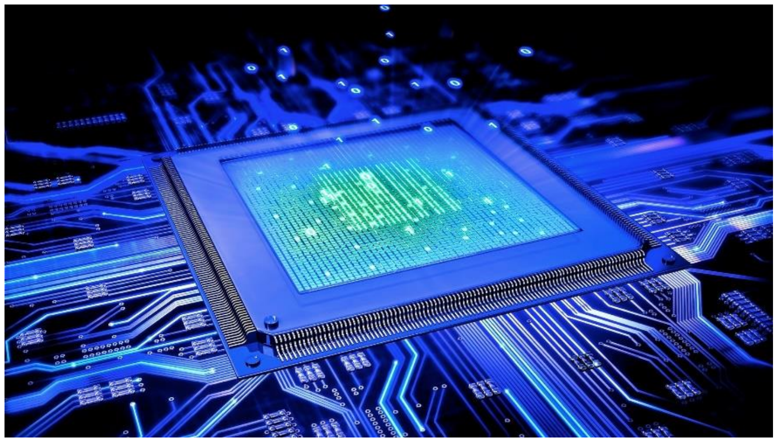

Homework #4

CPE 512

KYLE RAY

November 27, 2017

Contents

Part 1

ACTION 1.1

Table 1: Laplace 2D OpenMP Run Times with NP = 1,2,4,8

Laplace_2d_OpenMP	NP = 1	NP = 2	NP = 4	NP = 8
Run 1	113.3186	60.37949	36.38884	32.53741
Run 2	131.6103	72.48519	40.77835	26.132
Run 3	106.8418	57.42247	30.39545	22.02235
Run 4	131.4048	54.55593	62.06174	38.51126
Run 5	118.155	76.51001	45.77173	51.44997
Average (s)	120.2661	64.27062	43.07922	34.1306

ACTION 1.2

Table 2: Laplace 2D OpenAcc Run Times Using Kernels

Laplace_2d_OpenAcc	Run Time (s)
Run 1	209.357367
Run 2	201.063612
Run 3	215.540524
Run 4	214.036243
Run 5	215.446115
Average (s)	211.0887722

The average run time has increased dramatically using only the "#pragma acc kernels" on the main loops.

```
while (error > tol && iter < iter_max) {
  error = 0.0;

  #pragma acc kernels // Action 1.2 Change
  {
    for (int j = 1; j < n - 1; j++) {
      for (int i = 1; i < m - 1; i++) {
        Anew[j][i] = 0.25 * (A[j][i + 1] + A[j][i - 1]
          + A[j - 1][i] + A[j + 1][i]);
        error = fmax(error, fabs(Anew[j][i] - A[j][i]));
      }
    }

    for (int j = 1; j < n - 1; j++) {
      for (int i = 1; i < m - 1; i++) {
        A[j][i] = Anew[j][i];
      }
    }
  }


  if (iter % 100 == 0) cout << iter << "   " << setprecision(6)
    << fixed << error << endl;
  iter++;
}
```

The commented lines “Action 1.2 Change” are the portions of code that were changed for this action item.

ACTION 1.3

Table 3: Laplace 2D OpenAcc Run Times Using Loop Improvement

Laplace_2d_OpenAcc	Run Time (s)
Run 1	10.51169
Run 2	9.647732
Run 3	10.839015
Run 4	10.42466
Run 5	9.584973
Average (s)	10.201614

This is a significant improvement in execution time compared to the OpenMP version as well as just using the kernels directive in OpenAcc.

```
#pragma acc data copy(A) create(Anew) // Action 1.3 Change
{
  while (error > tol && iter < iter_max) {
    error = 0.0;

  #pragma acc kernels // Action 1.2 Change
    {
      for (int j = 1; j < n - 1; j++) {
        for (int i = 1; i < m - 1; i++) {
          Anew[j][i] = 0.25 * (A[j][i + 1] + A[j][i - 1]
            + A[j - 1][i] + A[j + 1][i]);
          error = fmax(error, fabs(Anew[j][i] - A[j][i]));
        }
      }

      for (int j = 1; j < n - 1; j++) {
        for (int i = 1; i < m - 1; i++) {
          A[j][i] = Anew[j][i];
        }
      }
    }


    if (iter % 100 == 0) cout << iter << "   " << setprecision(6)
      << fixed << error << endl;
    iter++;
  }
}
```

The commented line “Action 1.3 Change” are the portions of code that were changed for this action item.

ACTION 1.4

Table 4: Laplace 2D OpenAcc Run Times Loop Tuning

Laplace_2d_OpenAcc	Run Time (s)
Run 1	10.170871
Run 2	9.365193
Run 3	10.568781
Run 4	10.236471
Run 5	9.406877
Average (s)	9.9496386

This is a very significant improvement over the OpenMP implementation even when using 8 CPU cores.

```
#pragma acc data copy(A) create(Anew) // Action 1.3 Change
{
  while (error > tol && iter < iter_max) {
    error = 0.0;

    #pragma acc kernels // Action 1.2 Change
    {
      #pragma acc loop device_type(nvidia) tile(32,4) // Action 1.4 Change
      {
        for (int j = 1; j < n - 1; j++) {
          for (int i = 1; i < m - 1; i++) {
            Anew[j][i] = 0.25 * (A[j][i + 1] + A[j][i - 1]
              + A[j - 1][i] + A[j + 1][i]);
            error = fmax(error, fabs(Anew[j][i] - A[j][i]));
          }
        }
      }

      #pragma acc loop device_type(nvidia) tile(32,4) // Action 1.4 Change
      {
        for (int j = 1; j < n - 1; j++) {
          for (int i = 1; i < m - 1; i++) {
            A[j][i] = Anew[j][i];
          }
        }
      }
    }


    if (iter % 100 == 0) cout << iter << "  " << setprecision(6)
      << fixed << error << endl;
    iter++;
  }
}
```

ACTION 1.5

Table 5: Laplace 2D OpenAcc Run Times Parallel

Laplace_2d_OpenAcc	Run Time (s)
Run 1	176.446758
Run 2	221.737253
Run 3	215.029657
Run 4	281.299375
Run 5	275.355581
Average (s)	233.9737248

It seems that this is the slowest version yet, even slower than the OpenAcc implementation with just the kernels directive.

```
while (error > tol && iter < iter_max) {
  error = 0.0;

#pragma acc parallel loop reduction(max:error) device_type(nvidia) // Action 1.5 Change
  {
    for (int j = 1; j < n - 1; j++) {
      for (int i = 1; i < m - 1; i++) {
        Anew[j][i] = 0.25 * (A[j][i + 1] + A[j][i - 1]
          + A[j - 1][i] + A[j + 1][i]);
        error = fmax(error, fabs(Anew[j][i] - A[j][i]));
      }
    }
  }

#pragma acc parallel loop device_type(nvidia) // Action 1.5 Change
  {
    for (int j = 1; j < n - 1; j++) {
      for (int i = 1; i < m - 1; i++) {
        A[j][i] = Anew[j][i];
      }
    }
  }


  if (iter % 100 == 0) cout << iter << "  " << setprecision(6)
    << fixed << error << endl;
  iter++;
}
```

ACTION 1.6

Table 6: Laplace 2D OpenAcc Run Times Parallel Transfer Reduced

Laplace_2d_OpenAcc	Run Time (s)
Run 1	10.488272
Run 2	9.627119
Run 3	11.957242
Run 4	10.5251
Run 5	9.65133
Average (s)	10.4498126

Like before, when we cut out the many times the data had to be copied to the GPU, we have reached a significant performance increase. This is on par with the changes that were made in Action 1.3. Overall still a significant change from the OpenMP version.

```
#pragma acc data copy(A), create(Anew) // Action 1.6 Change
{
  while (error > tol && iter < iter_max) {
    error = 0.0;

  #pragma acc parallel loop reduction(max:error) device_type(nvidia) // Action 1.5 Change
    {
      for (int j = 1; j < n - 1; j++) {
        for (int i = 1; i < m - 1; i++) {
          Anew[j][i] = 0.25 * (A[j][i + 1] + A[j][i - 1]
            + A[j - 1][i] + A[j + 1][i]);
          error = fmax(error, fabs(Anew[j][i] - A[j][i]));
        }
      }
    }

  #pragma acc parallel loop device_type(nvidia) // Action 1.5 Change
    {
      for (int j = 1; j < n - 1; j++) {
        for (int i = 1; i < m - 1; i++) {
          A[j][i] = Anew[j][i];
        }
      }
    }


    if (iter % 100 == 0) cout << iter << "  " << setprecision(6)
      << fixed << error << endl;
    iter++;
  }
}
```

ACTION 1.7

Table 7: Laplace 2D OpenAcc Run Times Parallel Transfer Reduced Tuned

Laplace_2d_OpenAcc	Run Time (s)
Run 1	10.094672
Run 2	9.266023
Run 3	10.277465
Run 4	6.157706
Run 5	7.558197
Average (s)	8.6708126

Out of all the implementations thus far this is the lowest average execution time. With an average of 8.6 seconds over five runs compared to the 34 seconds over five runs for the OpenMP implementation utilizing 8 threads. The rest of the test cases with OpenMP do not even come close to the speed this solution offers. The GPU version seems to fair quite a bit better for this type of problem.

```
#pragma acc data copy(A), create(Anew) // Action 1.6 Change
{
  while (error > tol && iter < iter_max) {
    error = 0.0;

    #pragma acc parallel loop reduction(max:error) device_type(nvidia) tile (32, 4) //
Action 1.5 Change // Action 1.7 Change
    {
      for (int j = 1; j < n - 1; j++) {
        for (int i = 1; i < m - 1; i++) {
          Anew[j][i] = 0.25 * (A[j][i + 1] + A[j][i - 1]
            + A[j - 1][i] + A[j + 1][i]);
          error = fmax(error, fabs(Anew[j][i] - A[j][i]));
        }
      }
    }

    #pragma acc parallel loop device_type(nvidia) tile(32, 4) // Action 1.5 Change //
Action 1.7 Change
    {
      for (int j = 1; j < n - 1; j++) {
        for (int i = 1; i < m - 1; i++) {
          A[j][i] = Anew[j][i];
        }
      }
    }


    if (iter % 100 == 0) cout << iter << "  " << setprecision(6)
      << fixed << error << endl;
    iter++;
  }
}
```

Part 2

ACTION 2.1

Table 8: Matrix Multiplication Run Time for Square Matrix Size 8000 Serial

Matrix Multiplication	Matrix Size	Run Time (s)
Run 1	8000	6481.8289

Table 9: Matrix Multiplication Run Times for Square Matrix Size 8000 with OpenAcc

Matrix Multiplication	Matrix Size	Run Time (s)
Run 1	8000	15.85561
Run 2	8000	18.199035
Run 3	8000	20.671267
Run 4	8000	14.721847
Run 5	8000	15.83513
Average (s)		17.0565778

When testing the implementation with OpenAcc with smaller array sizes, just to check the output was correct, I was getting much larger execution times than the serial. I'm assuming this is because my implementation is copying the arrays over to the GPU and this incurs some overhead. For much larger array sizes, the tables turn, and the GPU is much faster than the serial version, over 6000 seconds vs. 17 seconds! The speedup that is gained from just adding a few lines of code to the serial implementation is astounding!

```
#pragma acc data copy(a[0:dim_l*dim_m]) copy(b[0:dim_m*dim_n]) copy(c[0:dim_l*dim_n])
{
  #pragma acc kernels
  {
    #pragma acc loop device_type(nvidia) tile(32,4)
    {
      for (i = 0; i < dim_l; i++) {
        for (j = 0; j < dim_n; j++) {
          dot_prod = 0.0;
          for (k = 0; k < dim_m; k++) {
            dot_prod += A(i, k)*B(k, j);
          }
          C(i, j) = dot_prod;
        }
      }
    }
  }
}
```

ACTION 2.2

Table 10: Matrix Multiplication Run Times for Various Square Matrix Sizes

Matrix Multiplication	Matrix Size	Serial (s)	MPI NT = 2	MPI NT = 4	MPI NT = 8	GPU
Test 1	512	0.818973	0.330414	0.155706	0.080532	5.284956
Test 2	1024	6.749437	3.352052	2.454554	1.240320	5.981198
Test 3	2048	91.993643	20.461957	12.809570	7.624370	6.019792
Test 4	4096	1294.302700	435.235120	337.036570	205.824700	7.565942

Looking at this we can see that the serial implementation execution time is growing exponentially with the size of the square matrices. While the MPI implementations are faster they are still growing exponentially just as the serially implementation. The GPU implementation, using OpenAcc however, seems to be increasing in a linear fashion from the results I've gathered out of these tests. This is an amazing decrease in execution time, and with a lot less work on the programmer! It is worthwhile to note that there seems to be a slight overhead encountered by copying the data over to the GPU. From other tests that I've ran while doing this assignment I was seeing that even with very small matrices that my OpenAcc implementation was running around 4-5 seconds. This can be seen in the 512 and 1024 sized matrix cases where even the serial implementation either is faster or is close to that of the GPU execution time. So, it would be safe to say that for very small matrices the OpenMPI versions would be better and for much larger matrices utilizing the GPU would be ideal.

Appendix

LAPLACE SOURCE CODE PART1 FINAL

```
/*
 *  Copyright 2017 NVIDIA Corporation
 *
 *  Licensed under the Apache License, Version 2.0 (the "License");
 *  you may not use this file except in compliance with the License.
 *  You may obtain a copy of the License at
 *
 *      http://www.apache.org/licenses/LICENSE-2.0
 *
 *  Unless required by applicable law or agreed to in writing, software
 *  distributed under the License is distributed on an "AS IS" BASIS,
 *  WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 *  See the License for the specific language governing permissions and
 *  limitations under the License.
   to load pgi compiler on dmc.asc.edu first type
      module load pgi

   to compile type
      pgc++ -fast laplace2d_acc.cpp -o laplace2d_acc  -acc -ta=nvidia -
Minfo=accel
   this creates the
      ./laplace2d_acc
   executable.

   to run -- use the run_gpu queue and place the following lines in a
             bash script file
      #!/bin/bash
     ./laplace2d_acc
*/

/*
   In Final Exam Problem 1 -- add the necessary OpenACC pragmas that are
                              needed to accomplish the goals of each part.
*/
using namespace std;
#include <iostream>
#include <iomanip>
#include <sstream>

#include <math.h>
#include <string.h>
#include "timer.h"

#define NN 4096
#define NM 4096

double A[NN][NM];
double Anew[NN][NM];

int main(int argc, char** argv)
{
   const int n = NN;
   const int m = NM;
   const int iter_max = 1000;

   const double tol = 1.0e-6;
   double error     = 1.0f;
```

```
    memset(A, 0, n * m * sizeof(double));
    memset(Anew, 0, n * m * sizeof(double));

    for (int j = 0; j < n; j++)
    {
        A[j][0]    = 1.0;
        Anew[j][0] = 1.0;
    }

    cout << "Jacobi relaxation Calculation: " << n << " x "
          << m << " mesh" << endl;

    StartTimer();
    int iter = 0;

    #pragma acc data copy(A) create(Anew) // Action 1.3 Change
    {
      while ( error > tol && iter < iter_max ) {
         error = 0.0;

         #pragma acc kernels // Action 1.2 Change
         {
           #pragma acc loop device_type(nvidia) tile(32,4) // Action 1.4 Change
           {
             for( int j = 1; j < n-1; j++) {
                 //#pragma acc loop gang(16), vector(32)
                 for( int i = 1; i < m-1; i++ ) {
                   Anew[j][i] = 0.25 * ( A[j][i+1] + A[j][i-1]
                              + A[j-1][i] + A[j+1][i]);
                   error = fmax( error, fabs(Anew[j][i] - A[j][i]));
                }
             }
           }

           #pragma acc loop device_type(nvidia) tile(32,4) // Action 1.4 Change
           {
             for( int j = 1; j < n-1; j++) {
                //#pragma acc loop gang(16), vector(32)
                for( int i = 1; i < m-1; i++ ) {
                   A[j][i] = Anew[j][i];
               }
             }
           }
         }


         if (iter % 100 == 0) cout << iter << "  " << setprecision(6)
                                   << fixed << error << endl;
         iter++;
      }
    }
    double runtime = GetTimer();
    cout << " total: " << runtime/1000 << " s" << endl;
}
```

LAPLACE SOURCE CODE PART 2 FINAL

```
/*
 *  Copyright 2017 NVIDIA Corporation
 *
 *  Licensed under the Apache License, Version 2.0 (the "License");
 *  you may not use this file except in compliance with the License.
 *  You may obtain a copy of the License at
 *
 *      http://www.apache.org/licenses/LICENSE-2.0
 *
 *  Unless required by applicable law or agreed to in writing, software
 *  distributed under the License is distributed on an "AS IS" BASIS,
 *  WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 *  See the License for the specific language governing permissions and
 *  limitations under the License.
   to load pgi compiler on dmc.asc.edu first type
      module load pgi

   to compile type
      pgc++ -fast laplace2d_acc.cpp -o laplace2d_acc  -acc -ta=nvidia -
Minfo=accel
   this creates the
      ./laplace2d_acc
   executable.

   to run -- use the run_gpu queue and place the following lines in a
             bash script file
      #!/bin/bash
     ./laplace2d_acc
*/

/*
   In Final Exam Problem 1 -- add the necessary OpenACC pragmas that are
                              needed to accomplish the goals of each part.
*/
using namespace std;
#include <iostream>
#include <iomanip>
#include <sstream>

#include <math.h>
#include <string.h>
#include "timer.h"

#define NN 4096
#define NM 4096

double A[NN][NM];
double Anew[NN][NM];

int main(int argc, char** argv)
{
   const int n = NN;
   const int m = NM;
   const int iter_max = 1000;

   const double tol = 1.0e-6;
   double error     = 1.0f;

   memset(A, 0, n * m * sizeof(double));
   memset(Anew, 0, n * m * sizeof(double));
   for (int j = 0; j < n; j++)
   {
       A[j][0]    = 1.0;
       Anew[j][0] = 1.0;
   }

   cout << "Jacobi relaxation Calculation: " << n << " x "
         << m << " mesh" << endl;

   StartTimer();
   int iter = 0;

   #pragma acc data copy(A), create(Anew) // Action 1.6 Change
   {
     while ( error > tol && iter < iter_max ) {
        error = 0.0;

        #pragma acc parallel loop reduction(max:error) device_type(nvidia) tile
(32, 4) // Action 1.5 Change // Action 1.7 Change
        {
          for( int j = 1; j < n-1; j++) {
             for( int i = 1; i < m-1; i++ ) {
                Anew[j][i] = 0.25 * ( A[j][i+1] + A[j][i-1]
                           + A[j-1][i] + A[j+1][i]);
                error = fmax( error, fabs(Anew[j][i] - A[j][i]));
             }
          }
        }

        #pragma acc parallel loop device_type(nvidia) tile(32, 4) // Action 1.5
Change // Action 1.7 Change
        {
          for( int j = 1; j < n-1; j++) {
             for( int i = 1; i < m-1; i++ ) {
                A[j][i] = Anew[j][i];
            }
          }
        }


        if (iter % 100 == 0) cout << iter << "  " << setprecision(6)
                                  << fixed << error << endl;
        iter++;
     }
   }

   double runtime = GetTimer();
   cout << " total: " << runtime/1000 << " s" << endl;
}
```

MATRIX MULTIPLICATION PART 2 OPENACC SOURCE CODE FINAL

```
/*****************************************************************/
/* Matrix Matrix Multiplication Program Example -- serial version */
/* September 2017 -- B. Earl Wells -- University of Alabama       */
/*                                    in Huntsville               */
/*****************************************************************/
// mm_mult_serial.cpp
// compilation:
//   gnu compiler
//      pgc++ mm_mult_serial.cpp -o mm_mult_serial -O3 -lm
/*
   This program is designed to perform matrix matrix multiplication
   A x B = C, where A is an lxm matrix, B is a m x n matrix and
   C is a l x n matrix. The program is designed to be a template
   serial program that can be expanded into a parallel multiprocess
   and/or a multi-threaded program.

   The program randomly assigns the elements of the A and B matrix
   with values between 0 and a MAX_VALUE. It then multiples the
   two matrices with the result being placed in the C matrix.
   The program prints out the A, B, and C matrices.

   The program is executed using one or three command line parameters.
   These parameters represent the dimension of the matrices. If only
   one parameter is used then then it is assumed that square matrices are
   to be created and multiplied together that have the specified
   dimension. In cases where three command line parameters are entered
   then the first parameter is the l dimension, the second the m, and
   the third is the n dimension.

   To execute:
   mm_mult_serial [l_parameter] <m_parameter n_parameter>
*/

using namespace std;
#include <iostream>
#include <iomanip>
#include <sstream>
#include <stdlib.h>
#include <string.h>
#include <sys/time.h>
#include "timer.h"

#define MX_SZ 320
#define SEED 2397           /* random number seed */
#define MAX_VALUE  100.0    /* maximum size of array elements A, and B */

/* copied from mpbench */
#define TIMER_CLEAR     (tv1.tv_sec = tv1.tv_usec = tv2.tv_sec = tv2.tv_usec =
0)
#define TIMER_START     gettimeofday(&tv1, (struct timezone*)0)
#define TIMER_ELAPSED   ((long long) (tv3.tv_usec)+((long long)
(tv3.tv_sec)*1000000))
#define TIMER_STOP      {gettimeofday(&tv2, (struct
timezone*)0);timersub(&tv2,&tv1,&tv3);}
struct timeval tv1,tv2,tv3;

/*
This declaration facilitates the creation of a two dimensional
dynamically allocated arrays (i.e. the lxm A array, the mxn B
array, and the lxn C array).  It allows pointer arithmetic to
```

```
be applied to a single data stream that can be dynamically allocated.
To address the element at row x, and column y you would use the
following notation:  A(x,y),B(x,y), or C(x,y), respectively.
Note that this differs from the normal C notation if A were a
two dimensional array of A[x][y] but is still very descriptive
of the data structure.
*/
float *a,*b,*c;
#define A(i,j) *(a+i*dim_m+j)
#define B(i,j) *(b+i*dim_n+j)
#define C(i,j) *(c+i*dim_n+j)

/*
   Routine to retrieve the data size of the numbers array from the
   command line or by prompting the user for the information
*/
void get_index_size(int argc,char *argv[],int *dim_l,int *dim_m,int *dim_n) {
   if(argc!=2 && argc!=4) {
      cout<<"usage:  mm_mult_serial [l_dimension] <m_dimension n_dimmension>"
           << endl;
      exit(1);
   }
   else {
      if (argc == 2) {
         *dim_l = *dim_n = *dim_m = atoi(argv[1]);
      }
      else {
         *dim_l = atoi(argv[1]);
         *dim_m = atoi(argv[2]);
         *dim_n = atoi(argv[3]);
      }
   }
   if (*dim_l<=0 || *dim_n<=0 || *dim_m<=0) {
      cout<<"Error: number of rows and/or columns must be greater than 0"
          << endl;
      exit(1);
   }
}

/*
   Routine that fills the number matrix with Random Data with values
   between 0 and MAX_VALUE
   This simulates in some way what might happen if there was a
   single sequential data acquisition source such as a single file
*/
void fill_matrix(float *array,int dim_m,int dim_n)
{
   int i,j;
   for(i=0;i<dim_m;i++) {
      for (j=0;j<dim_n;j++) {
         array[i*dim_n+j]=drand48()*MAX_VALUE;
      }
   }
}

/*
   Routine that outputs the matrices to the screen
*/
void print_matrix(float *array,int dim_m,int dim_n)
{
   int i,j;
   for(i=0;i<dim_m;i++) {
      for (j=0;j<dim_n;j++) {
         cout << array[i*dim_n+j] << " ";
      }
      cout << endl;
   }
}

/*
   MAIN ROUTINE: summation of a number list
*/

int main( int argc, char *argv[])
{
   float dot_prod;
   int dim_l,dim_n,dim_m;
   int i,j,k;
   /*
   get matrix sizes
   */
   get_index_size(argc,argv,&dim_l,&dim_m,&dim_n);

   // dynamically allocate from heap the numbers in the memory space
   // for the a,b, and c matrices
   a = new (nothrow) float[dim_l*dim_m];
   b = new (nothrow) float[dim_m*dim_n];
   c = new (nothrow) float[dim_l*dim_n];
   if(a==0 || b==0 || c==0) {
     cout <<"ERROR:  Insufficient Memory" << endl;
     exit(1);
   }

   /*
      initialize numbers matrix with random data
   */
   srand48(SEED);
   fill_matrix(a,dim_l,dim_m);
   fill_matrix(b,dim_m,dim_n);

   /*
     output numbers matrix
   */
   //cout << "A matrix =" << endl;
   //print_matrix(a,dim_l,dim_m);
   //cout << endl;

   //cout << "B matrix =" << endl;
   //print_matrix(b,dim_m,dim_n);
   //cout << endl;

   /*
   Start recording the execution time
   */
   //TIMER_CLEAR;
   //TIMER_START;
   //
   StartTimer();

   // multiply local part of matrix
   #pragma acc data copy(a[0:dim_l*dim_m]) copy( b[0:dim_m*dim_n])
copy(c[0:dim_l*dim_n])
   {
      #pragma acc kernels
      {
         #pragma acc loop device_type(nvidia) tile(32,4)
         {
            for (i=0;i<dim_l;i++) {
               for (j=0;j<dim_n;j++) {
                  dot_prod = 0.0;
                  for (k=0;k<dim_m;k++) {
                     dot_prod += A(i,k)*B(k,j);
                  }
                  C(i,j) = dot_prod;
               }
            }
         }
      }
   }
   /*
      stop recording the execution time
   */
   //TIMER_STOP;

   //cout << "C matrix =" << endl;
   //print_matrix(c,dim_l,dim_n);
   //cout << endl;
   double runTime = GetTimer();
   cout << "time=" << setprecision(8) <<  runTime/1000.0
        << " seconds" << endl;


}
```