Assignment 4

#Exercise 1:

```
#pragma scop
# for (i = 0; i < _PB_NI; i++) {
#   for (j = 0; j < _PB_NJ; j++)
#R:   C[i][j] *= beta;
#   for (k = 0; k < _PB_NK; k++) {
#     for (j = 0; j < _PB_NJ; j++)
#S:     C[i][j] += alpha * A[i][k] * B[k][j];
#   }
# }
#pragma endscop
print
"*************************************************************************************
***********";
```

#Question 1: Build the polyhedral representation (that is, write the ISCC script, as for assignment 2) of #the kernel above, that is the iteration domain of each statement, and each access function. Count how #many point are in each iteration domain.

#Solution:

```
print "Iteration Domain of R";
ID_R := [_PB_NI,_PB_NJ] -> {ID_R[i,j] : 0<=i<_PB_NI and 0<=j<_PB_NJ};
ID_R;
print "Cardinality of ID_R";
card ID_R;
print "Code for ID_R";
codegen ID_R;
print "Iteration Domain of S";
ID_S := [_PB_NI,_PB_NK,_PB_NJ] -> {ID_S[i,k,j] : 0<=i<_PB_NI and 0<=k<_PB_NK and 0<=j<_PB_NJ};
ID_S;
print "Cardinality of ID_S";
```

```
card ID_S;
print "Code for ID_S";
codegen ID_S;
print "Access function of R";
AF_RC := [_PB_NI,_PB_NJ] -> {ID_R[i,j] -> C[x][y] : x = i and y =j};
AF_RC;
print "Access function of S";
AF_SC := [_PB_NI,_PB_NK,_PB_NJ] -> {ID_S[i,j,k] -> C[x][y] : x = i and y = j};
AF_SC;
AF_SA := [_PB_NI,_PB_NK,_PB_NJ] -> {ID_S[i,j,k] -> A[x][z] : x = i and z = k};
AF_SA;
AF_SB := [_PB_NI,_PB_NK,_PB_NJ] -> {ID_S[i,j,k] -> B[z][y] : z = k and y = j};
AF_SB;
print
"*******************************************************************************
***********";
```

Output:

```
"*******************************************************************************
***********"
"Iteration Domain of R"
[_PB_NI, _PB_NJ] -> { ID_R[i, j] : 0 <= i < _PB_NI and 0 <= j < _PB_NJ }
"Cardinality of ID_R"
[_PB_NI, _PB_NJ] -> { _PB_NI * _PB_NJ : _PB_NI > 0 and _PB_NJ > 0 }
"Code for ID_R"
for (int c0 = 0; c0 < _PB_NI; c0 += 1)
  for (int c1 = 0; c1 < _PB_NJ; c1 += 1)
    ID_R(c0, c1);
"Iteration Domain of S"
[_PB_NI, _PB_NK, _PB_NJ] -> { ID_S[i, k, j] : 0 <= i < _PB_NI and 0 <= k < _PB_NK and 0 <= j < _PB_NJ }
"Cardinality of ID_S"
```

```
[_PB_NI, _PB_NK, _PB_NJ] -> { _PB_NI * _PB_NK * _PB_NJ : _PB_NI > 0 and _PB_NK > 0 and _PB_NJ > 0 }
"Code for ID_S"
for (int c0 = 0; c0 < _PB_NI; c0 += 1)
  for (int c1 = 0; c1 < _PB_NK; c1 += 1)
    for (int c2 = 0; c2 < _PB_NJ; c2 += 1)
      ID_S(c0, c1, c2);
"Access function of R"
[_PB_NI, _PB_NJ] -> { ID_R[i, j] -> C[i, j] }
"Access function of S"
[_PB_NI, _PB_NK, _PB_NJ] -> { ID_S[i, j, k] -> C[i, j] }
[_PB_NI, _PB_NK, _PB_NJ] -> { ID_S[i, j, k] -> A[i, k] }
[_PB_NI, _PB_NK, _PB_NJ] -> { ID_S[i, j, k] -> B[k, j] }
"**********************************************************************************
**********"
```

#Question 2: Compute the data space of the entire program using ISCC. Count how many points are in the data space.

#Solution:

```
print "Data space of C";
DS_C := AF_RC(ID_R) + AF_SC(ID_S);
DS_C;
print "Cardinality of DS_C";
card DS_C;
print "Code for DS_C";
codegen DS_C;
print "Data space of A";
DS_A := AF_SA(ID_S);
DS_A;
print "Cardinality of DS_A";
card DS_A;
```

```
print "Code for DS_A";

codegen DS_A;

print "Data space of B";

DS_B := AF_SB(ID_S);

DS_B;

print "Cardinality of DS_B";

card DS_B;

print "Code for DS_B";

codegen DS_B;

print "Dataspace of the entire kernel";

DS := DS_A + DS_B + DS_C;

DS;

print "Cardinality of DS";

card DS;

print "Code for DS";

codegen DS;

print "**********************************************************************************
**********";
```

Output:

```
"Data space of C"

[_PB_NI, _PB_NK, _PB_NJ] -> { C[x, y] : 0 <= x < _PB_NI and y >= 0 and (y < _PB_NJ or (_PB_NJ > 0 and y <
_PB_NK)) }

"Cardinality of DS_C"

[_PB_NI, _PB_NK, _PB_NJ] -> { _PB_NI * _PB_NK : _PB_NI > 0 and 0 < _PB_NJ < _PB_NK; _PB_NI * _PB_NJ
: _PB_NI > 0 and _PB_NJ >= _PB_NK and _PB_NJ > 0 }

"Code for DS_C"

if (_PB_NJ >= 1)

  for (int c0 = 0; c0 < _PB_NI; c0 += 1) {

    for (int c1 = 0; c1 < _PB_NJ; c1 += 1)
```

```
    C(c0, c1);
  for (int c1 = _PB_NJ; c1 < _PB_NK; c1 += 1)
    C(c0, c1);
 }
"Data space of A"
[_PB_NI, _PB_NK, _PB_NJ] -> { A[x, z] : _PB_NK > 0 and 0 <= x < _PB_NI and 0 <= z < _PB_NJ }
"Cardinality of DS_A"
[_PB_NI, _PB_NK, _PB_NJ] -> { _PB_NI * _PB_NJ : _PB_NI > 0 and _PB_NK > 0 and _PB_NJ > 0 }
"Code for DS_A"
if (_PB_NK >= 1)
  for (int c0 = 0; c0 < _PB_NI; c0 += 1)
    for (int c1 = 0; c1 < _PB_NJ; c1 += 1)
      A(c0, c1);
"Data space of B"
[_PB_NI, _PB_NK, _PB_NJ] -> { B[z, y] : _PB_NI > 0 and 0 <= z < _PB_NJ and 0 <= y < _PB_NK }
"Cardinality of DS_B"
[_PB_NI, _PB_NK, _PB_NJ] -> { _PB_NK * _PB_NJ : _PB_NI > 0 and _PB_NK > 0 and _PB_NJ > 0 }
"Code for DS_B"
if (_PB_NI >= 1)
  for (int c0 = 0; c0 < _PB_NJ; c0 += 1)
    for (int c1 = 0; c1 < _PB_NK; c1 += 1)
      B(c0, c1);
"Dataspace of the entire kernel"
[_PB_NI, _PB_NK, _PB_NJ] -> { B[z, y] : _PB_NI > 0 and 0 <= z < _PB_NJ and 0 <= y < _PB_NK; A[x, z] : _PB_NK > 0 and 0 <= x < _PB_NI and 0 <= z < _PB_NJ; C[x, y] : 0 <= x < _PB_NI and y >= 0 and (y < _PB_NJ or (_PB_NJ > 0 and y < _PB_NK)) }
"Cardinality of DS"
[_PB_NI, _PB_NK, _PB_NJ] -> { (_PB_NI * _PB_NK + (_PB_NI + _PB_NK) * _PB_NJ) : _PB_NI > 0 and _PB_NK > 0 and 0 < _PB_NJ < _PB_NK; (2 * _PB_NI + _PB_NK) * _PB_NJ : _PB_NI > 0 and _PB_NK > 0 and _PB_NJ >= _PB_NK and _PB_NJ > 0; _PB_NI * _PB_NJ : _PB_NI > 0 and _PB_NK <= 0 and _PB_NJ > 0 }
```

```
"Code for DS"
{
  if (_PB_NJ >= 1)
    for (int c0 = 0; c0 < _PB_NI; c0 += 1) {
      for (int c1 = 0; c1 < _PB_NJ; c1 += 1)
        C(c0, c1);
      for (int c1 = _PB_NJ; c1 < _PB_NK; c1 += 1)
        C(c0, c1);
    }
  if (_PB_NK >= 1)
    for (int c0 = 0; c0 < _PB_NI; c0 += 1)
      for (int c1 = 0; c1 < _PB_NJ; c1 += 1)
        A(c0, c1);
  if (_PB_NI >= 1)
    for (int c0 = 0; c0 < _PB_NJ; c0 += 1)
      for (int c1 = 0; c1 < _PB_NK; c1 += 1)
        B(c0, c1);
}
"*************************************************************************************"
```

#Question 3: Assume the following characteristics/values:
#- GPU: peak 1 TeraFlop/s, bandwidth RAM to/from GPU: 30GB/s
#- CPU: peak 100 GigaFlop/s, bandwidth RAM to/from CPU: 30GB/s
#- PB_NI = PB_NJ = PB_NK = 1000
#- data is floating point double-precision.
#Compute the minimal execution time of gemm on CPU and GPU using the roofline approach:
#minimal_exec_time = max(computation_time, communication_time), where:
#computation_time = (number of flops in the kernel) / (peak flop/s)
#communication_time = (number of bytes in the data space) / (bandwidth)
#Use the expressions from Questions 1 and 2 for the iteration domain sizes and data space size, #multiplying them ("manually", outside of ISCC) as needed to compute correctly the number of flops / #bytes in the data space, and minimal_exec_time for both CPU and GPU. What is the potential execution #time reduction possible by offloading this code to a GPU?

#Solution:

Number of flops in the kernel for CPU = 1000*1000+3*1000*1000*1000 = 3001000000 = 3.001Gflops
Peak flops/s for CPU = 100Gflops/s
Computation time for CPU = 3.001Gflops/100Gflops/s = 0.03001s = 30.01ms
Number of bytes in dataspace for CPU = 1000*1000 + (1000+1000)*1000 = 3000000 bytes = 3MB
Bandwidth for CPU = 30GB/s
Communication time for CPU = 3MB/30GB/s = 0.0001s = 0.1ms
Minimal_execution_time for CPU = max(30.1ms,0.1ms) = 30.1ms
Number of flops in the kernel for GPU = 1000*1000+3*1000*1000*1000 = 3001000000 = 3.001Gflops
Peak flops/s for CPU = 1Tflops/s
Computation time for CPU = 3.001Gflops/1Tflops/s = 0.003001s = 3.001ms
Number of bytes in dataspace for GPU = 1000*1000 + (1000+1000)*1000 = 3000000 bytes = 3MB
Bandwidth for CPU = 30GB/s
Communication time for CPU = 3MB/30GB/s = 0.0001s = 0.1ms
Minimal_execution_time for GPU= max(3.001ms,0.1ms) = 3.001ms
The potential reduction in execution time using GPU is by 27.0.99ms i.e. GPU is almost 10.03 time faster than CPU.

#Exercise 2:

#Question 1: Create the file gemm-simp.c by duplicating gemm.c, and changing the kernel to the #simplified one above. Then, produce a transformed C program (e.g., gemm-simp.pocc.c) by compiling #gemm-simp.c following the commands:

```
$> echo "4 4 4" > tile.sizes
$> ./bin/pocc --verbose --pluto-tile gemm-simp.c
$> rm tile.sizes
```

#Solution:

The transformed C program is given below:

```
#include <math.h>
/**
 * This version is stamped on May 10, 2016
 *
 * Contact:
 *   Louis-Noel Pouchet <pouchet.ohio-state.edu>
 *   Tomofumi Yuki <tomofumi.yuki.fr>
 *
 * Web address: http://polybench.sourceforge.net
 */
/* gemm.c: this file is part of PolyBench/C */
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <math.h>
/* Include polybench common header. */
#include <polybench.h>
/* Include benchmark-specific header. */
#include "gemm.h"
/* Array initialization. */
```

```
static void init_array(int ni, int nj, int nk, DATA_TYPE *alpha, DATA_TYPE *beta, DATA_TYPE POLYBENCH_2D(C,NI,NJ,ni,nj), DATA_TYPE POLYBENCH_2D(A,NI,NK,ni,nk), DATA_TYPE POLYBENCH_2D(B,NK,NJ,nk,nj))
{
  int i, j;
  *alpha = 1.5;
  *beta = 1.2;
  for (i = 0; i < ni; i++)
    for (j = 0; j < nj; j++)
      C[i][j] = (DATA_TYPE) ((i*j+1) % ni) / ni;
  for (i = 0; i < ni; i++)
    for (j = 0; j < nk; j++)
      A[i][j] = (DATA_TYPE) (i*(j+1) % nk) / nk;
  for (i = 0; i < nk; i++)
    for (j = 0; j < nj; j++)
      B[i][j] = (DATA_TYPE) (i*(j+2) % nj) / nj;
}
/* DCE code. Must scan the entire live-out data.
   Can be used also to check the correctness of the output. */
static void print_array(int ni, int nj, DATA_TYPE POLYBENCH_2D(C,NI,NJ,ni,nj))
{
  int i, j;
  POLYBENCH_DUMP_START;
  POLYBENCH_DUMP_BEGIN("C");
  for (i = 0; i < ni; i++)
    for (j = 0; j < nj; j++) {
          if ((i * ni + j) % 20 == 0) fprintf (POLYBENCH_DUMP_TARGET, "\n");
          fprintf (POLYBENCH_DUMP_TARGET, DATA_PRINTF_MODIFIER, C[i][j]);
    }
```

```
  POLYBENCH_DUMP_END("C");
  POLYBENCH_DUMP_FINISH;
}
/* Main computational kernel. The whole function will be timed,
   including the call and return. */
static void kernel_gemm(int ni, int nj, int nk, DATA_TYPE alpha, DATA_TYPE beta, DATA_TYPE POLYBENCH_2D(C,NI,NJ,ni,nj), DATA_TYPE POLYBENCH_2D(A,NI,NK,ni,nk), DATA_TYPE POLYBENCH_2D(B,NK,NJ,nk,nj))
{
  int i, j, k;
//BLAS PARAMS
//TRANSA = 'N'
//TRANSB = 'N'
// => Form C := alpha*A*B + beta*C,
//A is NIxNK
//B is NKxNJ
//C is NIxNJ
#ifdef ceild
# undef ceild
#endif
#ifdef floord
# undef floord
#endif
#ifdef max
# undef max
#endif
#ifdef min
# undef min
#endif
```

```
#define ceild(x,y) (((x) > 0)? (1 + ((x) - 1)/(y)): ((x) / (y)))
#define floord(x,y) (((x) > 0)? ((x)/(y)): 1 + (((x) -1)/ (y)))
#define max(x,y)    ((x) > (y)? (x) : (y))
#define min(x,y)    ((x) < (y)? (x) : (y))
/* Copyright (C) 1991-2016 Free Software Foundation, Inc.
   This file is part of the GNU C Library.
   The GNU C Library is free software; you can redistribute it and/or
   modify it under the terms of the GNU Lesser General Public
   License as published by the Free Software Foundation; either
   version 2.1 of the License, or (at your option) any later version.
   The GNU C Library is distributed in the hope that it will be useful,
   but WITHOUT ANY WARRANTY; without even the implied warranty of
   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
   Lesser General Public License for more details.
   You should have received a copy of the GNU Lesser General Public
   License along with the GNU C Library; if not, see
   <http://www.gnu.org/licenses/>.  */
/* This header is separate from features.h so that the compiler can
   include it implicitly at the start of every compilation.  It must
   not itself include <features.h> or any other header that includes
   <features.h> because the implicit include comes before any feature
   test macros that may be defined in a source file before it first
   explicitly includes a system header.  GCC knows the name of this
   header in order to preinclude it.  */
/* glibc's intent is to support the IEC 559 math functionality, real
   and complex.  If the GCC (4.9 and later) predefined macros
   specifying compiler intent are available, use them to determine
   whether the overall intent is to support these features; otherwise,
   presume an older compiler has intent to support these features and
```

```
   define these macros by default.  */
/* wchar_t uses Unicode 9.0.0.  Version 9.0 of the Unicode Standard is
   synchronized with ISO/IEC 10646:2014, fourth edition, plus
   Amd. 1  and Amd. 2 and 273 characters from forthcoming  10646, fifth edition.
   (Amd. 2 was published 2016-05-01,
   see https://www.iso.org/obp/ui/#iso:std:iso-iec:10646:ed-4:v1:amd:2:v1:en) */
/* We do not support C11 <threads.h>.  */
  register int lbv, ubv, lb, ub, lb1, ub1, lb2, ub2;
  register int c0, c1, c2, c3, c4, c5;
#pragma scop
if (((_PB_NI >= 1) && (_PB_NJ >= 1)) && (_PB_NK >= 1)) {
  for (c0 = 0; c0 <= floord((_PB_NI + -1), 4); c0++) {
    {
      for (c1 = 0; c1 <= floord((_PB_NJ + -1), 4); c1++) {
        {
          for (c2 = 0; c2 <= floord((_PB_NK + -1), 4); c2++) {
            {
              for (c3 = (4 * c0); c3 <= min((_PB_NI + -1), ((4 * c0) + 3)); c3++) {
                {
                  for (c4 = (4 * c1); c4 <= min((_PB_NJ + -1), ((4 * c1) + 3)); c4++) {
                    {
                      for (c5 = (4 * c2); c5 <= min((_PB_NK + -1), ((4 * c2) + 3)); c5++) {
                        {
                          C[c3][c4] += alpha * A[c3][c5] * B[c5][c4];
                        }
                      }
                    }
                  }
                }
              }
```

```
          }
         }
        }
       }
      }
     }
    }
  }
}
#pragma endscop
}
int main(int argc, char** argv)
{
  /* Retrieve problem size. */
  int ni = NI;
  int nj = NJ;
  int nk = NK;
  /* Variable declaration/allocation. */
  DATA_TYPE alpha;
  DATA_TYPE beta;
  POLYBENCH_2D_ARRAY_DECL(C,DATA_TYPE,NI,NJ,ni,nj);
  POLYBENCH_2D_ARRAY_DECL(A,DATA_TYPE,NI,NK,ni,nk);
  POLYBENCH_2D_ARRAY_DECL(B,DATA_TYPE,NK,NJ,nk,nj);
  /* Initialize array(s). */
  init_array (ni, nj, nk, &alpha, &beta, POLYBENCH_ARRAY(C), POLYBENCH_ARRAY(A), POLYBENCH_ARRAY(B));
  /* Start timer. */
  polybench_start_instruments;
  /* Run kernel. */
```

```
 kernel_gemm (ni, nj, nk, alpha, beta, POLYBENCH_ARRAY(C), POLYBENCH_ARRAY(A), POLYBENCH_ARRAY(B));

 /* Stop and print timer. */

 polybench_stop_instruments;

 polybench_print_instruments;

 /* Prevent dead-code elimination. All live-out data must be printed

   by the function call in argument. */

 polybench_prevent_dce(print_array(ni, nj,  POLYBENCH_ARRAY(C)));

 /* Be clean. */

 POLYBENCH_FREE_ARRAY(C);

 POLYBENCH_FREE_ARRAY(A);

 POLYBENCH_FREE_ARRAY(B);

 return 0;

}
```

#Question 2: For each of the 6 loops in the gemm-simp.pocc.c kernel, say if the loop is parallel (it is #not carrying any dependence) or if it is sequential. Beware that the above PoCC transformation #has changed the order of loops compared to the original code, to put parallel loops outer-most as #much as possible.

#Solution:

[Pluto] Outermost tilable band: c0--c2

[Pluto] After tiling:

c0 --> parallel tLoop (band 0) - parallel loop

c1 --> parallel tLoop (band 0) - parallel loop

c2 --> fwd_dep tLoop (band 0) - sequential loop

c3 --> parallel loop   (band 0) - parallel loop

c4 --> parallel loop   (band 0) - parallel loop

c5 --> fwd_dep loop   (band 0) - sequential loop

#Exercise 3:

#Question 1: Compute how many times the loop nest made of loops {c2,c3,c4,c5} is executed in the #program, using ISCC. Note you are now allowed to go beyond affine expressions, and use / (integer #division) and ceil/floor operators in the description of ISCC structures.

#Assuming NI=NJ=NK=1000, if we map each execution of {c2,c3,c4,c5} to a distinct CUDA thread, how #many #threads in total would be needed?

#print "Iteration Domain of c0 & c1";

#Solution:

```
ID_c0_c1 := [_PB_NI,_PB_NJ] -> {ID_c0_c1[i,j] : 0<=i<=floord((_PB_NI-1),4) and 0<=j<=floord((_PB_NJ-1),4)};

ID_c0_c1;

print "Cardinality of c0 & c1";

card ID_c0_c1;

print "**********************************************************************************************";
```

Output:

```
"Iteration Domain of c0 & c1"

[_PB_NI, _PB_NJ] -> { ID_c0_c1[i, j] : i >= 0 and 4i < _PB_NI and j >= 0 and 4j < _PB_NJ }

"Cardinality of c0 & c1"

[_PB_NI, _PB_NJ] -> { (floor((3 + _PB_NI)/4) * floor((3 + _PB_NJ)/4)) : _PB_NI > 0 and _PB_NJ > 0 }

Assuming _PB_NI =_PB_NJ =_PB_NK = 1000

Total threads = 62500
```

 # Question2: In CUDA programs, there is a hard limit on the number of threads per thread block, a #thread block cannot have more than 1024 threads. Let us assume that we use loop c0 to model #iterations over thread blocks, and loop c1 to model iterations over threads inside a block (i.e., for a #particular value of c0, that is a particular thread block). What is the range of values of PB_NI, PB_NJ #and PB_NK for gemm-simp.pocc.c which ensures we are not using more than 1024 threads per thread #block?

#Solution:

Since we use loop c0 to model iterations over thread blocks, and loop c1 to model iterations over threads inside a block i.e. the c0 is the number of blocks and c1 is the number of threads, _PB_NI can vary from 0 to 62500 and depending on _PB_NI, _PB_NJ and _PB_NK can vary from 0 to 1024 each to accommodate all 62500 threads.

#Question 3: We now map loop c0 to thread blocks, and loop c1 to threads inside a thread block, and #we assume PB_NI, PB_NJ and PB_NK are such that no more than 1024 threads per thread block is #needed/generated. Write the CUDA kernel function that executes {c2,c3,c4,c5,c6} in each thread, by #manually modifying the code to replace references to c0 and c1 by the adequate expressions involving #BlockDim.x, BlockIdx.x and ThreadId.x.

&

#Question 4: As a follow-up to Question 3, write the host code for this computation. Implement the #entire A, B and C matrices copy from host to GPU, and only the copy back of C from GPU to host. Refer #to the documents above for details.

#Note that you need to generate in the host code the call to your GPU kernel. In this mapping strategy, #we use 1D thread blocks. The number of blocks is given by the number of iterations of the loop c0, the #number of threads in a block is given by the number of iterations of the loop c1. So you should end up #with something similar to:

#Solution:

```
/**
 * This version is stamped on May 10, 2016
 *
 * Contact:
 *   Louis-Noel Pouchet <pouchet.ohio-state.edu>
 *   Tomofumi Yuki <tomofumi.yuki.fr>
 *
 * Web address: http://polybench.sourceforge.net
 */
/* gemm.c: this file is part of PolyBench/C */
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <math.h>
#include <sys/time.h>
#include <cuda_runtime.h>
/* Include polybench common header. */
#include <polybench.h>
```

```
/* Include benchmark-specific header. */
#include "gemm.h"
double cpuSecond()
{
        struct timeval tp;
        gettimeofday(&tp, NULL);
        return((double)tp.tv_sec + (double)tp.tv_usec*1e-6);
}
/* Array initialization. */
static void init_array(int ni, int nj, int nk,  DATA_TYPE *alpha, DATA_TYPE *beta, DATA_TYPE POLYBENCH_2D(C,NI,NJ,ni,nj), DATA_TYPE POLYBENCH_2D(A,NI,NK,ni,nk), DATA_TYPE POLYBENCH_2D(B,NK,NJ,nk,nj))
{
  int i, j;
  *alpha = 1.5;
  *beta = 1.2;
  for (i = 0; i < ni; i++)
    for (j = 0; j < nj; j++)
      C[i][j] = (DATA_TYPE) ((i*j+1) % ni) / ni;
  for (i = 0; i < ni; i++)
    for (j = 0; j < nk; j++)
      A[i][j] = (DATA_TYPE) (i*(j+1) % nk) / nk;
  for (i = 0; i < nk; i++)
    for (j = 0; j < nj; j++)
      B[i][j] = (DATA_TYPE) (i*(j+2) % nj) / nj;
}
/* DCE code. Must scan the entire live-out data.
   Can be used also to check the correctness of the output. */
static void print_array(int ni, int nj, DATA_TYPE POLYBENCH_2D(C,NI,NJ,ni,nj))
```

```
{
  int i, j;
  POLYBENCH_DUMP_START;
  POLYBENCH_DUMP_BEGIN("C");
  for (i = 0; i < ni; i++)
    for (j = 0; j < nj; j++) {
          if ((i * ni + j) % 20 == 0) fprintf (POLYBENCH_DUMP_TARGET, "\n");
          fprintf (POLYBENCH_DUMP_TARGET, DATA_PRINTF_MODIFIER, C[i][j]);
    }
  POLYBENCH_DUMP_END("C");
  POLYBENCH_DUMP_FINISH;
}
/* Main computational kernel. The whole function will be timed,
   including the call and return. */
static void kernel_gemm(int ni, int nj, int nk, DATA_TYPE alpha, DATA_TYPE beta, DATA_TYPE POLYBENCH_2D(C,NI,NJ,ni,nj), DATA_TYPE POLYBENCH_2D(A,NI,NK,ni,nk), DATA_TYPE POLYBENCH_2D(B,NK,NJ,nk,nj))
{
 // int i, j, k;
//BLAS PARAMS
//TRANSA = 'N'
//TRANSB = 'N'
// => Form C := alpha*A*B + beta*C,
//A is NIxNK
//B is NKxNJ
//C is NIxNJ
__global__ void matmul (int **a, int **b, int **c, int _PB_NI, int _PB_NJ, int _PB_NK, DATA_TYPE alpha) ;
#ifdef ceild
# undef ceild
```

```
#endif
#ifdef floord
# undef floord
#endif
#ifdef max
# undef max
#endif
#ifdef min
# undef min
#endif
#define ceild(x,y) (((x) > 0)? (1 + ((x) - 1)/(y)): ((x) / (y)))
#define floord(x,y) (((x) > 0)? ((x)/(y)): 1 + (((x) -1)/ (y)))
#define max(x,y)    ((x) > (y)? (x) : (y))
#define min(x,y)    ((x) < (y)? (x) : (y))
/* Copyright (C) 1991-2016 Free Software Foundation, Inc.
   This file is part of the GNU C Library.
   The GNU C Library is free software; you can redistribute it and/or
   modify it under the terms of the GNU Lesser General Public
   License as published by the Free Software Foundation; either
   version 2.1 of the License, or (at your option) any later version.
   The GNU C Library is distributed in the hope that it will be useful,
   but WITHOUT ANY WARRANTY; without even the implied warranty of
   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
   Lesser General Public License for more details.
   You should have received a copy of the GNU Lesser General Public
   License along with the GNU C Library; if not, see
   <http://www.gnu.org/licenses/>.  */
/* This header is separate from features.h so that the compiler can
   include it implicitly at the start of every compilation.  It must
```

```
   not itself include <features.h> or any other header that includes
   <features.h> because the implicit include comes before any feature
   test macros that may be defined in a source file before it first
   explicitly includes a system header.  GCC knows the name of this
   header in order to preinclude it.  */
/* glibc's intent is to support the IEC 559 math functionality, real
   and complex.  If the GCC (4.9 and later) predefined macros
   specifying compiler intent are available, use them to determine
   whether the overall intent is to support these features; otherwise,
   presume an older compiler has intent to support these features and
   define these macros by default.  */
/* wchar_t uses Unicode 9.0.0.  Version 9.0 of the Unicode Standard is
   synchronized with ISO/IEC 10646:2014, fourth edition, plus
   Amd. 1  and Amd. 2 and 273 characters from forthcoming  10646, fifth edition.
   (Amd. 2 was published 2016-05-01,
   see https://www.iso.org/obp/ui/#iso:std:iso-iec:10646:ed-4:v1:amd:2:v1:en) */
/* We do not support C11 <threads.h>.  */
//  register int lbv, ubv, lb, ub, lb1, ub1, lb2, ub2;
//  register int c0, c1, c2, c3, c4, c5;
            int **d_a, **d_b, **d_c;
            double iStart, iElaps;
            _PB_NI = 1024;
            _PB_NJ = 1024;
            _PB_NK = 1024;
            // initialize matrices a and b with appropriate values
            int size_A = _PB_NI * _PB_NI * sizeof(int);
            int size_B = _PB_NJ * _PB_NJ * sizeof(int);
            int size_C = _PB_NK * _PB_NK * sizeof(int);
            cudaMalloc((void **) &d_a, size_A);
```

```
	cudaMalloc((void **) &d_b, size_B);
	cudaMalloc((void **) &d_c, size_C);
	iStart = cpuSecond();
	cudaMemcpy(d_a, A, size_A, cudaMemcpyHostToDevice);
	cudaMemcpy(d_b, B, size_B, cudaMemcpyHostToDevice);
	dim3 dimGrid(60, 1);
	dim3 dimBlock(1024, 1024);
	matmul<<<dimGrid, dimBlock>>>(d_a, d_b, d_c, _PB_NI,_PB_NJ,_PB_NK,alpha);
	cudaMemcpy(C, d_c, size_C, cudaMemcpyDeviceToHost);
	cudaDeviceSynchronize();
	iElaps = cpuSecond() - iStart;
	printf("Time elapsed %f sec\n", iElaps);
	cudaFree(d_a);
	cudaFree(d_b);
	cudaFree(d_c);
}
__global__ void matmul (int **A, int **B, int **C, int _PB_NI, int _PB_NJ, int _PB_NK, DATA_TYPE alpha)
{
	int c2, c3, c4, c5 = 0;
	int col = threadIdx.x + blockDim.x * blockIdx.x;
	int row = threadIdx.y + blockDim.y * blockIdx.y;
	if (col < _PB_NI && row < _PB_NJ){
		for(c2 = 0; c2 <= floord((_PB_NK -1), 4); c2++ ){
			for (c3 = (4 * row); c3 <= min((_PB_NI + -1), ((4 * row) + 3)); c3++)
			{
			  for (c4 = (4 * col); c4 <= min((_PB_NJ + -1), ((4 * col) + 3)); c4++)
			  {
					for (c5 = (4 * c2); c5 <= min((_PB_NK + -1), ((4 * c2) + 3)); c5++)
					{
```

```
                                        C[c3][c4] += alpha * A[c3][c5] * B[c5][c4];
                                    }
                                }
                            }


                        }
                    }
}
int main(int argc, char** argv)
{
  /* Retrieve problem size. */
  int ni = NI;
  int nj = NJ;
  int nk = NK;
  /* Variable declaration/allocation. */
  DATA_TYPE alpha;
  DATA_TYPE beta;
  POLYBENCH_2D_ARRAY_DECL(C,DATA_TYPE,NI,NJ,ni,nj);
  POLYBENCH_2D_ARRAY_DECL(A,DATA_TYPE,NI,NK,ni,nk);
  POLYBENCH_2D_ARRAY_DECL(B,DATA_TYPE,NK,NJ,nk,nj);
  /* Initialize array(s). */
  init_array (ni, nj, nk, &alpha, &beta, POLYBENCH_ARRAY(C), POLYBENCH_ARRAY(A), POLYBENCH_ARRAY(B));
  /* Start timer. */
  polybench_start_instruments;
  /* Run kernel. */
  kernel_gemm (ni, nj, nk, alpha, beta, POLYBENCH_ARRAY(C), POLYBENCH_ARRAY(A), POLYBENCH_ARRAY(B));
  /* Stop and print timer. */
```

```
  polybench_stop_instruments;
  polybench_print_instruments;
  /* Prevent dead-code elimination. All live-out data must be printed
     by the function call in argument. */
  polybench_prevent_dce(print_array(ni, nj,  POLYBENCH_ARRAY(C)));
  /* Be clean. */
  POLYBENCH_FREE_ARRAY(C);
  POLYBENCH_FREE_ARRAY(A);
  POLYBENCH_FREE_ARRAY(B);
  return 0;
}
```

#Exercise 4:

#Question 1: Make all edits needed to obtain a stand-alone gemm-simp.pocc.cu file, which can compile #with nvcc. Likely if Exercise 3 was completed correctly this is only about renaming the file with the .cu #extension, but you may need to replace the various 'polybench_alloc_data' calls by a simple malloc() #on some setups if you see a compilation error related to 'posix_memalign' function being unavailable.

#Solution:

jhaveri@ip-172-31-81-235:~$ nvcc -O3 polybench-c-4.2.1-beta/linear-algebra/blas/gemm/gemm-simp.pocc.cu polybench-c-4.2.1-beta/utilities/polybench.c -Ipolybench-c-4.2.1-beta/utilities -DPOLYBENCH_TIME -o gemm-simp.transfo

jhaveri@ip-172-31-81-235:~$ ./gemm-simp.transfo

Time elapsed 0.000001 sec

#Question 2: We now assume PB_NI = PB_NJ = PB_NK = 1024. Compile and run the program for this #problem size, and report the execution time of the CUDA kernel. You can simply add a timer around #the cuda calls (from the first cudaMemcpy to the last one) or use nvprof. What is the GigaFlop/s #achieved by your implementation?

&

#Question 3: Taking the best performing version you obtained for Assignment 3, compute and report #its GigaFlop/s achieved, and compare it to what the GPU code achieved: is it faster?

#Solution:

Since we only have a compiler we cannot really launch our CUDA kernel. Hence we cannot calculate the execution time of CUDA kernel an hence, it is not possible to calculate the GigaFlop/s.

From Assignment 3, the best execution time achieved was 1.627871s for tile size 48, 72, 12 which is much greater than what we would achieve for a GPU . Hence a GigaFlops/s would be 1.979GFlops/s which would be much more in GPU. Hence GPU is much faster than CPU.