Λογισμικό και Προγραμματισμός Συστημάτων Υψηλής Επίδοσης

2016-2017

Εργασία Εξαμήνου

Ομάδα 11

Μενύχτας Βασίλειος, 4496
Μενύχτας Ευθύμιος, 5585

Εισαγωγή

Στα πλαίσια της εργασίας μας ζητήθηκε η υλοποίηση της πράξης του πολλαπλασιασμού A^TA με χρήση της CUDA, για γενικά μητρώα Α. Δημιουργήσαμε τρεις διαφορετικές υλοποιήσεις όπως ζητήθηκε, μια με χρήση της έτοιμης συνάρτησης cublasDgemm, μια με απλό πολλαπλασιασμό στα πρότυπα της CUDA, και μια όπου συμπεριλάβαμε τις όσες βελτιστοποιήσεις μπορέσαμε να εντοπίσουμε, για την καλύτερη αξιοποίηση του hardware. Οι κώδικες των υλοποιήσεων μας περιλαμβάνονται στα αρχεία cublas-mm.cu, parallel-basic.cu και parallel-optimized.cu αντίστοιχα.

Σε κάθε υλοποίηση η αρχικοποίηση του Α πραγματοποιείται στον host με χρήση της συνάρτησης rand(), η οποία γίνεται seed με την time(NULL), εντός δυο εμφωλευμένων for loops. Η μεταφορά του μητρώου Α στη καθολική μνήμη της κάρτας γραφικών γίνεται με τη συνάρτηση cublasSetMatrix() για τη περίπτωση της cublasDgemm ενώ για τις άλλες δυο γίνεται με τη cudaMemcpy(). Για κάθε υλοποίηση η μεταφορά των αποτελεσμάτων απ'το device στο host γίνεται με τη cudaMemcpy(). Επίσης, η μέτρηση των χρόνων σε κάθε περίπτωση γίνεται με χρήση των cudaEvents και της cudaEventRecord().

cuBLAS

Για την υλοποίηση της cuBLAS εκτελέσαμε την cublasDgemm() με τα παρακάτω ορίσματα και σύμφωνα με τις οδηγίες του documentation:

```
cublasDgemm(handle,opN,opT,cols,cols,rows,&alpha,dA,cols,dA,cols,&beta,dC,cols);
```

Όπως αναφέρεται στο documentation η cuBLAS αποθηκεύει τα μητρώα με column-major format, επομένως όταν περνάμε στη μνήμη της GPU το μητρώο Α, αυτό αποθηκεύεται έτσι ώστε αν το προσπελαύναμε με row-major λογική, θα διαβάζαμε το Α ανάστροφο. Βάση αυτού λοιπόν, οι παράμετροι δώθηκαν ως εξής:

`transa = opN` → έχει οριστεί ως CUBLAS_OP_N , διατηρεί το Α ως έχει (A^T σε row-major)

`transb = opT` → έχει οριστεί ως CUBLAS_OP_T , κάνει transpose το Α (οπότε Α σε row-major)

`m = cols` → έχει οριστεί ως το πλήθος στηλών του Α, και ισούται με το ύψος του C

`n = cols` → έχει οριστεί ως το πλήθος γραμμών του A^T και ισούται με το πλάτος του C

`k = rows` → πλήθος γραμμών του Α/στηλών του A^T

`alpha = &alpha` → έχει οριστεί με τιμή 1

`A = dA` → δείκτης για το Α στο device

`lda = cols` → η πρωτεύουσα διάσταση του Α (στήλες διότι έχουμε column-major format)

`B = dA` → επίσης δείκτης για το A στο device

`ldb = cols` → ίδιο όπως πριν

`beta = &beta` → έχει οριστεί με τιμή 0

`C = dC` → δείκτης για το C στο device

`ldc = cols` → η πρωτεύουσα διάσταση του C

Οι χρόνοι που πήραμε από την εκτέλεση του κώδικα στη Tesla C2075 παρουσιάζονται στο πίνακα παρακάτω. Οι τιμές είναι σε msec.

A size	2000^2	4000^2	6000^2	8000^2	10000^2	12000^2	14000^2	16000^2	18000^2
Time	51.95	408.34	1481.28	3458.49	6773.91	11702.40	18660.41	27671.52	39509.19

Απλός παράλληλος A^TA

Ο κώδικας της συνάρτησης πυρήνα για την απλή υλοποίηση της ζητούμενης πράξης φαίνεται παρακάτω.

```
__global__ void matrixMultiply(double* mA, double* mC, int rows, int
cols){
  int k;

  int row = blockIdx.y*blockDim.y + threadIdx.y;
  int col = blockIdx.x*blockDim.x + threadIdx.x;

  double val = 0.0;
  if ((row < cols) && (col < cols)){
    for (k = 0; k < rows; k++){
      val += mA[k*cols + row] * mA[k*cols + col];
    }
    mC[row*cols + col] = val;
  }
}
```

Για την περίπτωση αυτή λοιπόν, ορίζουμε τις row και col μεταβλητές που αποτελούν τις συντεταγμένες του εκάστοτε thread μέσα στο grid και με την if επιλέγουμε τα threads που δε ξεφεύγουν των ορίων του μητρώου mC, στο οποίο θέλουμε να αποθηκεύσουμε τα αποτελέσματα. Κάθε thread θα υπολογίσει το στοιχείο που αντιστοιχεί στις συντεταγμένες του εντός του mC πολλαπλασιάζοντας την υπ'αριθμόν row γραμμή του A^T με την υπ'αριθμόν col στήλη του A. Εφόσον στη μνήμη της GPU έχουμε μόνο το μητρώο A και λαμβάνοντας υπ'όψιν πως η i-οστή γραμμή του A^T είναι η i-οστή στήλη του A, βάζουμε το thread να προσπελάσει τη γραμμή σαν στήλη του A. Έτσι, με χρήση της for, κάθε thread θα διανύσει τις row και col στήλες του A, πολλαπλασιάζοντας κάθε φορά τα στοιχεία μεταξύ τους και προσθέτοντας το αποτέλεσμα στο γενικό άθροισμα για τον υπολογισμό του στοιχείου $c_{row, col}$.

Ακολουθεί πίνακας με τις μετρήσεις που πήραμε για το παραπάνω κώδικα για μέγεθος block 16x16.

A size	2000^2	4000^2	6000^2	8000^2	10000^2	12000^2	14000^2	16000^2	18000^2
Time	429.29	3482.16	12008.3	29365.12	58601.13	105631.83	170900.72	268574.53	378438.31

Βελτιστοποιημένος A^TA

Στο σημείο αυτό θα περιγράψουμε τη λογική που ακολουθήσαμε μέχρι το τελικό κώδικα για το βελτιστοποιημένο πολλαπλασιασμό.

Σε πρώτο στάδιο αποφασίσαμε να προσεγγίσουμε το πρόβλημα με tiles προκειμένου να αξιοποιήσουμε τη shared memory. Ορίσαμε το μέγεθος για το tile με μια σταθερά TILE_WIDTH και στη συνέχεια δημιουργήσαμε δυο shared πίνακες δυο διαστάσεων και τύπου double όπου θα περνάνε οι τιμές απ'το A. Ο καινούριος μας kernel είναι ο ακόλουθος:

```
__global__ void matrixMultiply(double* mA, double* mC, int rows, int cols)
{
  __shared__ double sT[TILE_WIDTH][TILE_WIDTH];
  __shared__ double sN[TILE_WIDTH][TILE_WIDTH];
  int k, m;

  int tx = threadIdx.x;
  int ty = threadIdx.y;
  int row = blockIdx.y*TILE_WIDTH + ty;
  int col = blockIdx.x*TILE_WIDTH + tx;

  double tval = 0.0;

  for (k = 0; k < (rows - 1) / TILE_WIDTH + 1; k++) {
    if ((row < cols) && (k*TILE_WIDTH+tx < rows)) {
      sT[ty][tx] = mA[(k*TILE_WIDTH+tx)*cols + row];
    } else {
      sT[ty][tx] = 0.0;
    }
    if ((col < cols) && (k*TILE_WIDTH+ty < rows)) {
      sN[ty][tx] = mA[(k*TILE_WIDTH+ty)*cols + col];
    } else {
      sN[ty][tx] = 0.0;
    }
    __syncthreads();

    for (m = 0; m < TILE_WIDTH; m++) {
      tval += sT[ty][m] * sN[m][tx];
    }
    __syncthreads();
  }

  if ((row < cols) && (col < cols)) {
    mC[row*cols + col] = tval;
  }
}
```

Με την μετατροπή σε tiled προσπέλαση του μητρώου αναγκαζόμαστε να εισάγουμε καινούριες συνθήκες. Πλέον δεν τρέχει το κάθε thread κάτα μήκος των στηλών με βήμα 1 και για όσο είναι το ύψος του A, αλλά τρέχει το tile κάτα μήκος του A καλύπτοντας TILE_WIDTH πλήθος στηλών με βήμα TILE_WIDTH. Ωστόσο για λόγους ευκολίας αντί να μεταβάλουμε το βήμα της for, αλλάξαμε κατάλληλα τη συνθήκη τερματισμού της σε (rows – 1)/TILE_WIDTH +1. Εντός του tile τώρα, κάθε thread αναλαμβάνει σε κάθε βήμα να διαβάσει δυο τιμές του πίνακα A και να τις μεταφέρει στους αντίστοιχους shared υποπίνακες των A^T και A. Επειδή το μέγεθος του πίνακα A όμως δεν είναι πολλαπλάσιο του TILE_WIDTH, οφείλουμε να σιγουρευτούμε ότι δεν θα περάσουν στη κοινή μας μνήμη τιμές από τη global memory που βρίσκονται σε θέσεις μνήμης παρακείμενες του πίνακα A. Για το λόγο αυτό χρησιμοποιούμε

τις συνθήκες if-else όπως φαίνονται παραπάνω. Με τους περιορισμούς row < cols και col < cols εξασφαλίζουμε πως δε ξεπερνάμε το A σε πλάτος ενώ με τα k*TILE_WIDTH+tx και k*TILE_WIDTH+ty πως δε το ξεπερνάμε σε ύψος. Για τις περιπτώσεις που βγαίνουμε εκτός ορίων του A, εισάγουμε ως τιμή το 0 στο αντίστοιχο στοιχείο του shared πίνακα.

Στη συνέχεια, μετά τη syncthreads όπου βεβαιώνουμε ότι οι shared πίνακες έχουν γεμίσει με τα κατάλληλα στοιχεία, κάθε thread διανύει το tile, όπως ακριβώς θα διένυε τον A στον βασικό αλγόριθμο, πολλαπλασιάζοντας τις τιμές των δυο διανυσμάτων που του αντιστοιχούν και αθροίζοντας τα αποτελέσματα. Κάθε φορά που μετακινείται το tile, το άθροισμα αυξάνεται έως ότου το thread θα έχει υπολογίσει τη τελική τιμή, την οποία περνά στο τέλος στο πίνακα mC.

Παρατηρούμε πως έχουμε δυο if-else συνθήκες εντός της for loop οι οποίες προκαλούν thread divergence. Ένας τρόπος που μπορούμε να μειώσουμε το φαινόμενο είναι ο ακόλουθος:

```
__global__ void matrixMultiply(double* mA, double* mC, int rows, int cols)
{
  __shared__ double sT[TILE_WIDTH][TILE_WIDTH];
  __shared__ double sN[TILE_WIDTH][TILE_WIDTH];
  double T, N;
  int k, m;

  int tx = threadIdx.x;
  int ty = threadIdx.y;
  int row = blockIdx.y*TILE_WIDTH + ty;
  int col = blockIdx.x*TILE_WIDTH + tx;

  double tval = 0.0;

  for (k = 0; k < gridDim.y; k++) {
    T = 0.0;
    N = 0.0;
    if ((row < cols) && (k*TILE_WIDTH+tx < rows)) {
      T = mA[(k*TILE_WIDTH+tx)*cols + row];
    }
    if ((col < cols) && (k*TILE_WIDTH+ty < rows)) {
      N = mA[(k*TILE_WIDTH+ty)*cols + col];
    }
    sT[ty][tx] = T;
    sN[ty][tx] = N;
    __syncthreads();

    for (m = 0; m < TILE_WIDTH; m++) {
      tval += sT[ty][m] * sN[m][tx];
    }
    __syncthreads();
  }
  if ((row < cols) && (col < cols)) {
    mC[row*cols + col] = tval;
  }
}
```

Εισάγαμε τις δυο μεταβλητές T και N, τις οποίες αρχικοποιούμε με μηδέν σε κάθε βήμα της for. Σε περίπτωση που ικανοποιείται κάποια απ’τις δυο if, η τιμή της κάθε μεταβλητής

ανανεώνεται με μια ανάγνωση από τη global ενώ στη συνέχεια η τιμή περνάει στη shared memory.

Στη συνέχεια, επιλέξαμε σαν TILE_WIDTH το 16. Με αυτή τη τιμή το πλήθος των threads σε κάθε block είναι 256, ενώ το μέγεθος της shared memory που χρησιμοποιεί κάθε block διαμορφώνεται στα 4096 byte [(256 στοιχεία) x (8 byte το sizeof(double)) x (2 πίνακες)] το οποίο μας δίνει μέγιστο occupancy για τους SMs της κάρτας μας. Επιπλέον, ως πολλαπλάσιο του 32, οι μεταφορές που γίνονται από τη global memory με το κάθε warp είναι coalesced.

Έχοντας ορίσει σταθερό TILE_WIDTH 16, έχουμε τη δυνατότητα να μειώσουμε το instruction overhead που προσθέτει η εμφωλευμένη for. Εφόσον το TILE_WIDTH δεν είναι ιδιαίτερα μεγάλο, μπορούμε να τη ξεδιπλώσουμε ως εξής:

```
val += sT[ty][15] * sN[15][tx];
val += sT[ty][14] * sN[14][tx];
val += sT[ty][13] * sN[13][tx];
val += sT[ty][12] * sN[12][tx];
val += sT[ty][11] * sN[11][tx];
val += sT[ty][10] * sN[10][tx];
val += sT[ty][9] * sN[9][tx];
val += sT[ty][8] * sN[8][tx];
val += sT[ty][7] * sN[7][tx];
val += sT[ty][6] * sN[6][tx];
val += sT[ty][5] * sN[5][tx];
val += sT[ty][4] * sN[4][tx];
val += sT[ty][3] * sN[3][tx];
val += sT[ty][2] * sN[2][tx];
val += sT[ty][1] * sN[1][tx];
val += sT[ty][0] * sN[0][tx];
```

Μια επιπλέον βελτιστοποίηση κρύβεται στον αριθμό των πράξεων που εκτελούμε. Εφόσον πολλαπλασιάζουμε το A ανάστροφο με το A, το μητρώο που θα προκύψει εκτός από τετραγωνικό θα είναι και συμμετρικό. Επομένως, μπορούμε να εκτελέσουμε τους υπολογισμούς για το πάνω τριγωνικό μέρος του μητρώου και να αντιγράψουμε τα αποτελέσματα στις αντίστοιχες θέσεις του κάτω τριγώνου όταν θα επιστρέφουμε στο πίνακα mC τα αποτελέσματα. Δηλαδή, εισάγουμε τις παραπάνω πράξεις μέσα σε μια if και τροποποιούμε τις συνθήκες για την εγραφή στο mC ως εξής:

```
if (row <= col) {
   val += sT[ty][15] * sN[15][tx];
   val += sT[ty][14] * sN[14][tx];

     ……
     ……
     ……

   val += sT[ty][1] * sN[1][tx];
   val += sT[ty][0] * sN[0][tx];
}
__syncthreads();
```

και

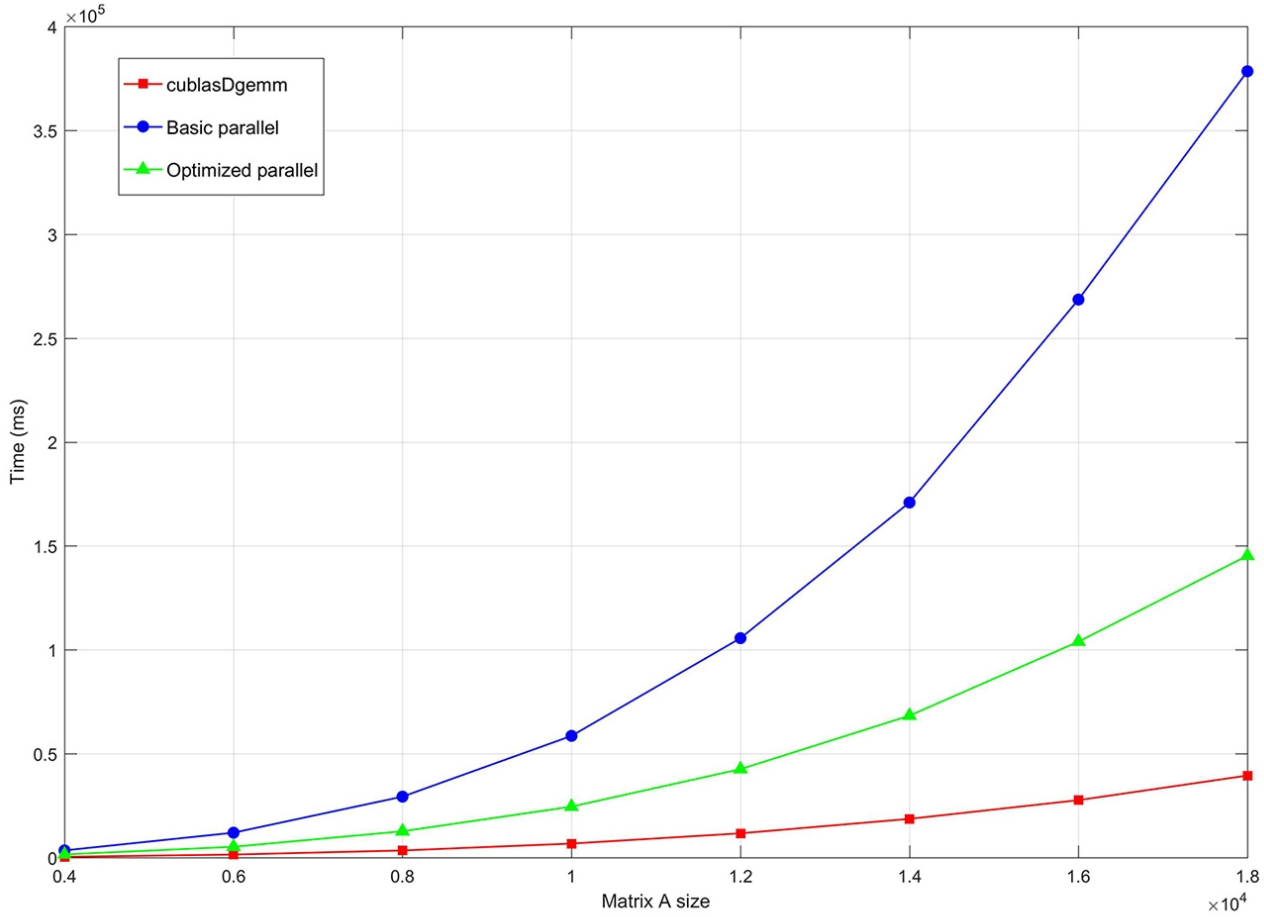
```
if ((row <= col) && (col < cols)) {
    mC[row*cols + col] = val;
```

```
        if (row != col) {
            mC[col*cols + row] = val;
        }
    }
```

Ακολουθεί πίνακας με τις μετρήσεις για τη τελική έκδοση της βελτιστοποιημένης υλοποιήσης.

A size	2000^2	4000^2	6000^2	8000^2	10000^2	12000^2	14000^2	16000^2	18000^2
Time	191.41	1536.83	5226.62	12690.03	24552.65	42585.66	68320.30	103951.38	145295.08

Στο παρακάτω γράφημα παρουσιάζονται οι μετρήσεις και των τριών αλγορίθμων για τα διάφορα μεγέθη μητρώων που δοκιμάσαμε.



Τέλος παραθέτουμε τον ολοκληρωμένο κώδικα της βελτιστοποιημένης έκδοσης μας ο οποίος περιλαμβάνεται και στο αρχείο parallel-optimized.cu. Οι διαστάσεις του μητρώου Α δίνονται σαν παράμετροι στο terminal κατά τη κλήση του εκτελέσιμου.

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <time.h>
#include <cuda_runtime.h>
#include <cuda.h>

#define TILE_WIDTH 16

void cudaErrorCheck() {
     cudaError_t code = cudaGetLastError();
     if (code != cudaSuccess) {
          printf("[CUDA error]: %s\n", cudaGetErrorString(code));
          exit(1);
     }
}

__global__ void matrixMultiply(double* mA, double* mC, int rows, int cols){
     __shared__ double sT[TILE_WIDTH][TILE_WIDTH];
     __shared__ double sN[TILE_WIDTH][TILE_WIDTH];
     double T, N;
     int k;

     int tx = threadIdx.x;
     int ty = threadIdx.y;
     int row = blockIdx.y*blockDim.y + ty;
     int col = blockIdx.x*blockDim.x + tx;

     double val = 0.0;
     for (k = 0; k < gridDim.y; k++) {
          T = 0.0; N = 0.0;
          if ((row < cols) && (k*TILE_WIDTH + tx < rows)) {
               T = mA[(k*TILE_WIDTH + tx)*cols + row];
          }
          if ((col < cols) && (k*TILE_WIDTH + ty < rows)) {
               N = mA[(k*TILE_WIDTH + ty)*cols + col];
          }
          sT[ty][tx] = T;
          sN[ty][tx] = N;
          __syncthreads();

          if (row <= col) {
               val += sT[ty][15] * sN[15][tx];
               val += sT[ty][14] * sN[14][tx];
               val += sT[ty][13] * sN[13][tx];
               val += sT[ty][12] * sN[12][tx];
               val += sT[ty][11] * sN[11][tx];
               val += sT[ty][10] * sN[10][tx];
               val += sT[ty][9] * sN[9][tx];
               val += sT[ty][8] * sN[8][tx];
               val += sT[ty][7] * sN[7][tx];
               val += sT[ty][6] * sN[6][tx];
               val += sT[ty][5] * sN[5][tx];
               val += sT[ty][4] * sN[4][tx];
               val += sT[ty][3] * sN[3][tx];
               val += sT[ty][2] * sN[2][tx];
               val += sT[ty][1] * sN[1][tx];
               val += sT[ty][0] * sN[0][tx];
          }
```

```
            __syncthreads();
        }

        if ((row <= col) && (col < cols)) {
            mC[row*cols + col] = val;
            if (row != col) {
                mC[col*cols + row] = val;
            }
        }
}

int main(int argc, char *argv[]){
        int i, j, pr_dev = 0;

        if (argv[1] == NULL){
            printf("Matrix dimensions not specified.\n\n");
            exit(1);
        } else if (argv[2] == NULL) {
            printf("Assuming square matrix.\n\n");
            argv[2] = argv[1];
        } else if (strcmp(argv[2], "-pd") == 0) {
            printf("Assuming square matrix.\n\n");
            argv[2] = argv[1];
            pr_dev = 1;
        } else if (argv[3] != NULL) {
            if (strcmp(argv[3], "-pd") == 0) {
            pr_dev = 1;
            }
        }
        int rows = atoi(argv[1]); // A transposed width
        int cols = atoi(argv[2]); // A transposed height -> Dimensions of result C
        /* Allocation and initialization of matrices on host ---------------------*/
        double *hA, *hC;

        hA = (double *)malloc(cols*rows*sizeof(double));
        hC = (double *)malloc(cols*cols*sizeof(double));
        srand(time(NULL));
        for (i = 0; i < rows; i++){
            for (j = 0; j < cols; j++){
                hA[i*cols + j] = rand() / 1000000.0;
            }
        }
        /* Allocation of matrices on the GPU -----------------------------------*/
        double *dA, *dC;

        cudaMalloc((void **)&dA, cols*rows*sizeof(double));
        cudaMalloc((void **)&dC, cols*cols*sizeof(double));
        cudaMemcpy(dA, hA, rows*cols*sizeof(double), cudaMemcpyHostToDevice);
        /* Get device properties & print them if asked --------------------------*/
        cudaDeviceProp props;
        cudaGetDeviceProperties(&props, 0);

        if (pr_dev == 1) {
            printf("   [ Device :: %s ]\n", props.name);
            printf("  Number of SMs ---------------: %d\n", props.multiProcessorCount);
            printf("  Global memory ---------------: %lu MB\n",
props.totalGlobalMem/1048576);
            printf(" Constant memory -------------: %lu KB\n", props.totalConstMem/1024);
            printf("  Threads per warp ------------: %d\n", props.warpSize);
```

```
          printf("  Max threads per block -------: %d\n", props.maxThreadsPerBlock);
          printf("  Max registers per block -----: %dK\n", props.regsPerBlock/1024);
          printf("  Max shared memory per block -: %lu KB\n",
props.sharedMemPerBlock/1024);
          printf("  Max block dimension ---------: %d x %d x %d\n",
props.maxThreadsDim[0], props.maxThreadsDim[1], props.maxThreadsDim[2]);
          printf("  Max grid dimension ----------: %d x %d x %d\n\n",
props.maxGridSize[0], props.maxGridSize[1], props.maxGridSize[2]);
     }
     /* Kernel invocation & computation timing -------------------------------*/
     unsigned int block_width = TILE_WIDTH;
     unsigned int grid_dim = (cols - 1) / block_width + 1;
     dim3 dimGrid(grid_dim, grid_dim, 1);
     dim3 dimBlock(block_width, block_width, 1);

     printf("Matrix A ---: %d x %d\n", rows, cols);
     printf("Grid size --: %u x %u\n", grid_dim, grid_dim);
     printf("Block size -: %u x %u\n\n", block_width, block_width);

     cudaEvent_t com_begin, com_end, mem_begin, mem_end;
     cudaEventCreate(&com_begin); cudaEventCreate(&com_end);
     cudaEventCreate(&mem_begin); cudaEventCreate(&mem_end);

     cudaEventRecord(com_begin, 0);
     matrixMultiply<<<dimGrid , dimBlock>>>(dA, dC, rows, cols);
     cudaEventRecord(com_end, 0);
     cudaEventSynchronize(com_end);

     cudaEventRecord(mem_begin, 0);
     cudaMemcpy(hC, dC, cols*cols*sizeof(double), cudaMemcpyDeviceToHost);
     cudaEventRecord(mem_end, 0);
     cudaEventSynchronize(mem_end);

     float com_time = 0;
     float mem_time = 0;
     cudaEventElapsedTime(&com_time, com_begin, com_end);
     cudaEventElapsedTime(&mem_time, mem_begin, mem_end);
     printf("Computation time: %.3f ms\n", com_time);
     printf("Memory transfer: %.3f ms\n", mem_time);
     /* Print output to file for computation evaluation ----------------------*/
     if (cols*rows <= 4096){
          FILE *f = fopen("data.txt", "w");
          if (f == NULL) {
               printf("Error: failed to open file.\n");
               exit(1);
          }
          for (i = 0; i < cols; i++){
               for (j = 0; j < cols; j++){
                    fprintf(f, "%f ", hC[i*cols + j]);
               }
               fprintf(f, ";\n");
          }
     }

     free(hA); cudaFree(dA);
     free(hC); cudaFree(dC);
     cudaErrorCheck();
     return 0;
}
```