MULTIPROCESSOR PROGRAMMING, DV2597/DV2606
—
LAB 2: GPU PROGRAMMING

Sai Prashanth Josyula, Håkan Grahn, Lars Lundberg
Blekinge Institute of Technology

2021 fall
uppdated: 2021-11-11

The objective of this laboratory assignment is to introduce basic GPU programming, and give some experience of how work and data partitioning as well as communication and synchronization impact the performance of parallel applications on a GPU.

The laboratory assignments should be conducted, solved, implemented, and presented in groups of two students! It is ok to do them individually, but groups of more than two students are not allowed.

Home Assignment 1. Prerequisties and preparations

You are expected to have acquired the following knowledge before the lab session:

- You are supposed to have good programming experience and good knowledge of C / C++ programming.
- You are supposed to have basic knowledge about concepts related to GPU programming.
- You should have installed a GPU programming platform (preferably CUDA) on the computer where you intend to perform the tasks.

End of home assignment 1.

Home Assignment 2. Plagiarism and collaboration

You are encouraged to work in groups of two. Groups larger than two are not accepted.

Discussions between laboratory groups are positive and can be fruitful. It is normally not a problem, but watch out so that you do not cross the border to cheating. For example, you are *not allowed* to share solution approaches, solutions to the different tasks, source code, output data, results, etc.

The submitted solution(s) to the laboratory assignments and tasks, should be developed by the group members only. You are not allowed to copy code from somewhere else, i.e., neither from your student fellows nor from the internet or any other source. The only other source code that is allowed to use is the one provided with the laboratory assignment.

End of home assignment 2.

1 Introduction

In this laboratory assignment, we will work with graphics processing units (GPUs). The lab is divided into two GPU programming assignments, **both of which need to be submitted**:

- An assignment that involves the use of basic concepts and basic programming of GPU (Section 3.1). This assignment is mainly intended to get you started with GPU programming and to understand how synchronization works.

- An assignment covering how to parallelize an algorithm to achieve high performance (Section 3.2).

In this lab, the goal is to parallelize and optimize the programs so they efficiently utilize the available graphics processing unit. The C and C++ programs in this lab assignment are tested on Kraken. However, they should also work on the computers in the lab room as well as your personal computers. The source code files and programs necessary for the laboratory are available on Canvas. You shall compile the programs provided to you using:

```
gcc -O2 -o output_filename sourcefile.c
```

```
g++ -std=c++0x -O2 -o output_filename sourcefile.cpp
```

Compiling a CUDA program is similar to compiling the above C and C++ programs. A CUDA compiler called `nvcc` is included in the CUDA toolkit that you should have installed on your computer. Using this compiler, you can compile your CUDA code (in your .cu file) as follows:

```
nvcc -o output_filename sourcefile.cu
```

2 Examination and grading

Note: Present and discuss your solutions with a teacher / lab assistant when all tasks are done, i.e., before submitting on Canvas.

When you have discussed your solutions orally, prepare and submit a `tar`-file or `zip`-file containing:

- **Source code:** The source-code for working solutions to the tasks in Sections 3.1 and 3.2. Specifically, you shall submit your well-commented source code for **Tasks 1, 2**, and **3**.
- Corresponding `Makefile`(s), or a text-file describing how to compile the respective projects / tasks.
- **Written report:** You should write a short report (approximately 2-3 pages) describing, *for each task*, your implementations, the answers to the questions in the tasks, and your measurements (results), as outlined below. The format of the report must be pdf.
  - **Implementation:** A general description of your parallel implementations, i.e., you should describe how you have partitioned the work between the GPU threads, how the data structures are organized, how the GPU threads are synchronized within a block and across blocks, etc.
  - **Measurements:** You should provide execution times for two cases: (i) the sequential version of the application, and (ii) the parallel version of the application running on a GPU (using as many threads as you like).
    The data set sizes for the tasks are: Odd-even sort using an array with 2^{19} items, and Gauss-Jordan row reduction using a 2048×2048 matrix.
    Further, analyze and discuss your results.

All material (except the code given to you in this assignment) must be produced by the laboratory group alone.

The examiner may contact you within a week, if they need some oral clarifications on your code or report. In this case, all group members must be present at that oral occasion.

3 Assignments

This section presents the tasks to be solved and submitted for this lab. You are going to parallelize two sequential algorithms using CUDA, i.e., Odd-even sort (Section 3.1) and Gauss-Jordan row reduction (Section 3.2).

3.1 Odd-even transposition sorting

The problem to be solved in this task is to implement a parallel version of *Odd-even sort* using CUDA. This is a simple sorting algorithm, originally developed for using on a parallel architecture. The C++ code for a sequential implementation is found in Appendix A (Listing 1, `oddevensort.cpp`). The provided code generates a sequence of pseudorandom integers and also times a sequential version of the odd-even sort. You are expected to implement an odd-even sorting kernel. Record and compare the time used for sorting using your CUDA program with our odd-even sort program.

The odd-even sort algorithm is presented as Algorithm 9.3 in [2]. In this algorithm, shown in Figure 1, $a_0, a_1, a_2, \dots a_{n-1}$ is the list of n elements that are to be sorted. A sorted output is guaranteed after n iterations of the `for` loop. Note that the input may already be fully sorted in an i^{th} iteration of the `for` loop, where $i < n$. The algorithm could be written in a way such that it checks during every `for` loop iteration whether the input is fully sorted or not. However, we do not deal with that version of the algorithm in this task.

```
1.    procedure ODD-EVEN($n$)
2.    begin
3.        for $i := 1$ to $n$ do
4.        begin
5.            if $i$ is odd then
6.                for $j := 0$ to $n/2 - 1$ do
7.                    compare-exchange($a_{2j+1}, a_{2j+2}$);
8.            if $i$ is even then
9.                for $j := 1$ to $n/2 - 1$ do
10.                   compare-exchange($a_{2j}, a_{2j+1}$);
11.       end for
12.   end ODD-EVEN
```

Figure 1: Sequential odd-even transposition sort algorithm (Algorithm 9.3 in [2]).

Task 1. Parallel Odd-even sort using a single kernel launch

Write a parallel implementation of Odd-even sort using CUDA. You will need to perform synchronization after each odd/even phase. In your kernel, synchronize the GPU threads using `__syncthreads()`. After you launch your kernel for a given input, you should see that it will sort correctly only when it is launched with a single block of threads.

Measure and compare the execution times for (i) the sequential version given to you, (ii) your parallel version launched with a single block of threads[a], and calculate the speedup of your parallel application.

End of task 1.

[a]you may launch up to 2048 threads in a block, depending on your hardware

Task 2. Parallel Odd-even sort using multiple kernel launches

Extend your previous CUDA implementation of the Odd-even sort such that it works for any number of thread blocks. Remember that by using multiple kernel launches, you can synchronize threads over the entire grid and not only within the same block (as `__syncthreads()` does). Your updated kernel should now sort correctly for any launch configuration.

Measure and compare the execution times for (i) the sequential version given to you, (ii) your updated parallel version running with as many threads as you like. Further, calculate the speedup of your parallel application.

End of task 2.

3.2 Gauss-Jordan row reduction method

The Gauss-Jordan row reduction method is an extension of the Gaussian elimination method [1]. In the Gaussian elimination method, we solved a given linear system of equations (let's say the unknown variables are $x_1, x_2 \dots x_n$) by performing computations on matrices. In that method, we proceeded through the columns from left to right, creating pivots, and eliminating elements below the pivots [1] (see Figure 2). The matrices resulting from the Gaussian elimination method gave us the value of x_n. When using Gaussian elimination, we need to perform a *backward substitution* step to get the values of the remaining variables $x_1, x_2, \dots x_{n-1}$ that satisfy the given system of equations. However, we can extend the Gaussian elimination method to also zero out the entries above each pivot, as we proceed from column to column [1]. By doing this, the values of $x_1, x_2, \dots x_n$ can be obtained without explicitly doing a backward substitution. This extended version of the Gaussian elimination is called the Gauss-Jordan row reduction method [1].

The problem to be solved in this task is to implement a parallel version of Gauss-Jordan row reduction using CUDA. The Gaussian elimination algorithm is described in [2] (presented as Algorithm 8.4). You shall initialize the matrix `A`, and the vectors `y` and `b` with reasonable values (done by default). This algorithm can be extended to the Gauss-Jordan row reduction algorithm by adding the pseudo code shown in Figure 3. The code for a sequential implementation of Gauss-Jordan row reduction is found in Appendix A (Listing 2).

```
1.   procedure GAUSSIAN_ELIMINATION ($A, b, y$)
2.   begin
3.       for $k := 0$ to $n - 1$ do              /* Outer loop */
4.       begin
5.           for $j := k + 1$ to $n - 1$ do
6.               $A[k, j] := A[k, j]/A[k, k]$;   /* Division step */
7.           $y[k] := b[k]/A[k, k]$;
8.           $A[k, k] := 1$;
9.           for $i := k + 1$ to $n - 1$ do
10.          begin
11.              for $j := k + 1$ to $n - 1$ do
12.                  $A[i, j] := A[i, j] - A[i, k] \times A[k, j]$; /* Elimination step */
13.              $b[i] := b[i] - A[i, k] \times y[k]$;
14.              $A[i, k] := 0$;
15.          endfor;           /* Line 9 */
16.      endfor;               /* Line 3 */
17.  end GAUSSIAN_ELIMINATION
```

Figure 2: Serial Gaussian elimination algorithm (Algorithm 8.4 in [2]).

```
1. for $i := 0$ to $k-1$ do
2. begin
3.     for $j := k + 1$ to $n - 1$ do
4.         $A[i, j] := A[i, j] - A[i, k] \times A[k, j]$;    /* Another elimination step */
5.     $y[i] := y[i] - A[i, k] \times y[k]$;
6.     $A[i, k] := 0$
7. endfor;
```

Figure 3: Pseudocode to include between lines 15 and 16 of the algorithm in Figure 2 to achieve Gauss-Jordan row reduction.

Task 3. Parallel Gauss-Jordan row reduction

You are supposed to do the following:

- Write a parallel implementation of Gauss-Jordan row reduction using CUDA.
- Measure the speedup of your parallel version on a GPU using a 2048×2048 matrix.

End of task 3.

References

[1] S. F. Andrilli, and D. Hecker, “Elementary linear algebra, 4th ed.,” *Academic Press*, 2010, ISBN: 978-0-12-374751-8.

[2] A. Grama, A. Gupta, G. Karypis, and V. Kumar, “Introduction to Parallel Computing, 2nd ed.,” *Addison-Wesley*, 2003, ISBN 0-201-64865-2.

[3] T. Rauber and G. Rünger, “Parallel Programming for Multicore and Cluster Systems, 2nd ed.,” *Springer*, 2013, ISBN 978-3-642-37800-3.

A Program listings

Listing 1. `oddevensort.cpp`

```
#include <vector>
#include <algorithm>
#include <iostream>
#include <chrono>

// The odd−even sort algorithm
// Total number of odd phases + even phases = the number of elements to sort
void oddeven_sort(std::vector<int>& numbers)
{
    auto s = numbers.size();
    for (int i = 1; i <= s; i++) {
        for (int j = i % 2; j < s; j = j + 2) {
            if (numbers[j] > numbers[j + 1]) {
                std::swap(numbers[j], numbers[j + 1]);
            }
        }
    }
}

void print_sort_status(std::vector<int> numbers)
{
    std::cout << "The input is sorted?: " << (std::is_sorted(numbers.begin(), numbers.end()) == 0 ? "False" : "True") << std::endl;
}

int main()
{
    constexpr unsigned int size = 100000; // Number of elements in the input

    // Initialize a vector with integers of value 0
    std::vector<int> numbers(size);
    // Populate our vector with (pseudo)random numbers
    srand(time(0));
    std::generate(numbers.begin(), numbers.end(), rand);

    print_sort_status(numbers);
    auto start = std::chrono::steady_clock::now();
    oddeven_sort(numbers);
    auto end = std::chrono::steady_clock::now();
    print_sort_status(numbers);
    std::cout << "Elapsed time = " << std::chrono::duration<double>(end − start).count() << " sec\n";
}
```

Listing 2. `gaussjordanseq.c`

```
/***************************************************************************
 *
 * Sequential version of Gauss−Jordan row reduction
 *
 ***************************************************************************/

#include <stdio.h>

#define MAX_SIZE 4096

typedef double matrix[MAX_SIZE][MAX_SIZE];

int N;  /* matrix size  */
int maxnum;  /* max number of element*/
char* Init;  /* matrix init type */
```

```
int PRINT;  /* print switch  */
matrix A;  /* matrix A  */
double b[MAX_SIZE]; /* vector b */
double y[MAX_SIZE]; /* vector y */

/* forward declarations */
void work(void);
void Init_Matrix(void);
void Print_Matrix(void);
void Init_Default(void);
int Read_Options(int, char**);

int
main(int argc, char** argv)
{
    printf("Gauss Jordan\n");
    int i, timestart, timeend, iter;

    Init_Default();  /* Init default values */
    Read_Options(argc, argv); /* Read arguments */
    Init_Matrix();  /* Init the matrix */
    work();
    if (PRINT == 1)
        Print_Matrix();
}

void
work(void)
{
    int i, j, k;

    /* Gaussian elimination algorithm, Algo 8.4 from Grama */
    for (k = 0; k < N; k++) { /* Outer loop */
        for (j = k + 1; j < N; j++)
            A[k][j] = A[k][j] / A[k][k]; /* Division step */
        y[k] = b[k] / A[k][k];
        A[k][k] = 1.0;
        for (i = k + 1; i < N; i++) {
            for (j = k + 1; j < N; j++)
                A[i][j] = A[i][j] - A[i][k] * A[k][j]; /* Elimination step */
            b[i] = b[i] - A[i][k] * y[k];
            A[i][k] = 0.0;
        }
        for (i = 0; i < k; i++) {
            for (j = k + 1; j < N; j++)
                A[i][j] = A[i][j] - A[i][k] * A[k][j]; /* Additional Elimination for Gauss-Jordan */
            y[i] = y[i] - A[i][k] * y[k];
            A[i][k] = 0.0;
        }
    }
}

void
Init_Matrix()
{
    int i, j;

    printf("\nsize = %dx%d ", N, N);
    printf("\nmaxnum = %d \n", maxnum);
    printf("Init = %s \n", Init);
    printf("Initializing matrix...");

    if (strcmp(Init, "rand") == 0) {
        for (i = 0; i < N; i++) {
            for (j = 0; j < N; j++) {
                if (i == j) /* diagonal dominance */
                    A[i][j] = (double)(rand() % maxnum) + 5.0;
                else
                    A[i][j] = (double)(rand() % maxnum) + 1.0;
            }
        }
    }
    if (strcmp(Init, "fast") == 0) {
        for (i = 0; i < N; i++) {
            for (j = 0; j < N; j++) {
                if (i == j) /* diagonal dominance */
                    A[i][j] = 5.0;
                else
                    A[i][j] = 2.0;
            }
        }
    }

    /* Initialize vectors b and y */
    for (i = 0; i < N; i++) {
        b[i] = 2.0;
        y[i] = 1.0;
    }

    printf("done \n\n");
    if (PRINT == 1)
        Print_Matrix();
}

void
Print_Matrix()
{
    int i, j;

    printf("Matrix A:\n");
    for (i = 0; i < N; i++) {
        printf("[");
        for (j = 0; j < N; j++)
            printf(" %5.2f,", A[i][j]);
        printf("]\n");
    }
    printf("Vector y:\n[");
    for (j = 0; j < N; j++)
        printf(" %5.2f,", y[j]);
    printf("]\n");
    printf("\n\n");
}

void
Init_Default()
{
    N = 2048;
    Init = "fast";
    maxnum = 15.0;
    PRINT = 0;
}

int
Read_Options(int argc, char** argv)
{
    char* prog;

    prog = *argv;
    while (++argv, --argc > 0)
        if (**argv == '-')
            switch (*++ * argv) {
            case 'n':
                --argc;
                N = atoi(*++argv);
                break;
            case 'h':
                printf("\nHELP: try sor -u \n\n");
                exit(0);
                break;
            case 'u':
                printf("\nUsage: gaussian [-n problemsize]\n");
                printf(" [-D] show default values \n");
                printf(" [-h] help \n");
                printf(" [-I init_type] fast/rand \n");
```

```
                    printf(" [−m maxnum] max random no \n");
                    printf(" [−P print_switch] 0/1 \n");
                    exit(0);
                    break;
               case 'D':
                    printf("\nDefault: n = %d ", N);
                    printf("\n Init = rand");
                    printf("\n maxnum = 5 ");
                    printf("\n P = 0 \n\n");
                    exit(0);
                    break;
               case 'I':
                    −−argc;
                    Init = *++argv;
                    break;
               case 'm':
                    −−argc;
                    maxnum = atoi(*++argv);
                    break;
               case 'P':
                    −−argc;
                    PRINT = atoi(*++argv);
                    break;
               default:
                    printf("%s: ignored option: −%s\n", prog, *argv);
                    printf("HELP: try %s −u \n\n", prog);
                    break;
               }
}
```