Programming Assignment 2
CSC410 - Parallel Computing

Aaron G. Alphonsus

November 14, 2018

1 Matrix-Vector Multiplication

A matrix-vector multiplication takes an input Matrix **A**, and a vector **B** and produces an output vector **C**. Each element of the output vector **C** is the dot product of one row of the input Matrix **A** and vector **B**, that is, `C[i] =` $\sum_j$ `A[i][j] · B[j]`.

This program contains the host program and CUDA kernel to multiply a square matrix with a vector.

2 Performance Analysis

In order to do the performance analysis, we used NVIDIA's profiling tool `nvprof`. This gave us the time taken by the kernel function as well as different CUDA API calls.

The Karp-Flatt metric provides a means to use empirical data to learn something about parallel overhead as well as the amount of inherently sequential computation in a parallel program. The basic idea is to collect data from several runs of the program with increasing numbers of processors.

Definition: Given a parallel computation showing a speedup $\psi(n, p)$ on p processors, where $p > 1$, the experimentally determined serial fraction f_e is defined to be:

$$f_e = \frac{1/\psi(n,p) - 1/p}{1 - 1/p}$$

The speedup $\psi(n, p)$ is calculated as usual - the ratio between sequential execution time and parallel execution time.

$$\psi(n,p) = \frac{T(n,1)}{T(n,p)}$$

With these formulas, we can benchmark our program. We assumed number of threads to be equal to p. We used a block size of 256 and increased number of blocks by 1 repeatedly. This is why the increase in p is by 256 each time:

p	*256*	*512*	*768*	*1024*	*1280*	*1536*	*1792*	*2048*
$\psi(n,p)$	92.95	175.93	169.81	224.40	203.94	239.84	224.87	256.82
f_e	0.0069	0.0037	0.0046	0.0035	0.0041	0.0035	0.0039	0.0034

What we can see here is that the serial fraction is flattening out indicating that there is limited opportunity for more parallelism. We also notice that some grid size / block size configurations are better than others in achieving speedup even though the total number of threads may be less. We hypothesize that this is due to the design of the hardware.

3 Algorithms and Libraries

Our program follows the typical algorithm for a CUDA C program with a few differences:

1. Declare and allocate host and device memory (we allocated unified memory for the vectors).
2. Initialize vectors on the host. For a quick and easy check of our calculations, we initialized each row of **A** from `1` to `n` and the vector **B** from `1` to `n`. This means that each position in **C** is the sum of squares from `1` to `n` is given by the formula:

$$\frac{n(n+1)(2n+1)}{6}$$

3. Transfer vectors from the host to the device. (This is a usual step i CUDA programs but since we declared unified memory, we did not have do this).
4. Execute the matrix-vector multiplication kernel. We went with coarse-grained parallelism with each thread multiplying a row of matrix **A** with the vector **B**. The dot product summation was done serially.
5. Transfer results from device to host (Not required since we used unified memory).
6. Free the unified memory.

Libraries used:

- The CUDA platform and API
- `<stdio.h>`

4 Functions and Program Structure

The program has 2 functions:

- `main`
- `matvecMul`

4.1 `main`

Arguments: none

Returns: `0` indicating normal termination.

Description:

- Declares vectors and allocates unified memory to them.
- Initializes the vector **B** and each row of the matrix **A** with numbers from `1` to `n`.
- Calls the CUDA kernel after defining grid and block sizes to execute the multiplication in parallel.
- Prints out resultant vector **C**.

4.2 `matvecMul`

Arguments:

- `double *A`: Square matrix **A** of dimension `n * n`.
- `double *B`: Column vector **B** of length `n`.
- `double *C`: Result of matrix-vector multiplication, row vector **C** of length `n`.
- `int n`: Dimension variable for the matrix and vector (set to be 8192)

Returns: void

Description:

- CUDA kernel code that executes the matrix-vector multiplication on the GPU.
- Coarse-grained parallelism: threads run in parallel for each row of the matrix **A** but execute the dot-product multiplication serially.

5 Compilation and Usage

Compilation: `nvcc -o prog2 prog2.cu` OR `make`
Usage: `./prog2`

The program can be compiled using the command `make`. To get rid of the executable in the folder, run the command `make clean`.

The program multiplies an `n x n` matrix with a vector of length `n`. The vector and each row of the matrix is filled with numbers from `1` to `n`. The resulting vector is the sum of squares from `1` to `n` in each position. `n` is initialized as `8192`.

6 Testing and Verification

For verification as we built the program, we used small values of `n` and printed out all three vectors. The answers of the matrix multiplication were verified using Maple. An example can be seen in figure 1.

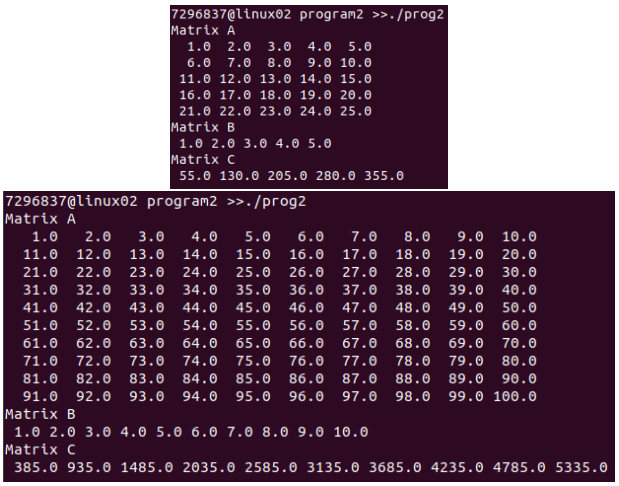

Figure 1: Comparison of the static and dynamic schedulers

Once we had a correct solution, we wanted a quick way to test that the program continued to work as we increased the number of elements. To do this we initialized the vector and each row of the matrix with numbers from `1` to `n`. We did this so that we would have a closed form solution for the resulting vector (sum of squares).
We then calculate the sum of squares, compare it with each element in our resultant matrix, count the mismatches, and print the count and the sum of squares to the screen. The resultant matrix is also printed to the screen along with the grid size and block size for verification.

7 Files Submitted

- `prog2.pdf`
- `Makefile`
- `prog2.cu`
- `latex/`