CpSc 418
33 points

Homework 4

Due: March 9, 2018, 11:59pm
Early Bird: March 7, 2018, 11:59pm

Template and support code for the assignment can be found at

http://www.ugrad.cs.ubc.ca/~cs418/2017-2/hw/4/code.html.

Tests included with the template code are not complete, and you should expect that we will run additional tests against your code.

Please submit your solution using the handin program. Submit your solution as

```
cs418 hw4
```

Your submission should consist of the following files:

- `hw4.erl`: Your solution to the key cracking problem.
- `recurr-kernel.cu`: Your solution to the getting started with CUDA problem.
- `hw4.txt` or `hw4.pdf`: Written response portion of the getting started with CUDA problem.

These files must be at the top level of your submitted archive and **not** in a subdirectory. Any other files submitted will be ignored, including any other support code files from the assignment template—your submission files must compile and run with the unmodified support files.

Please submit code that compiles without errors or warnings. If your code does not compile, we might give you zero points on the corresponding programming problem. If we fix your code to make it compile, we will take off lots of points for that service. If your code generates compiler warnings, we will take off points for that as well, but not as many as for code that doesn't compile successfully.

We will take off points for code that prints results unless we specifically asked for a print-out or the template code we provided generates it. Using `format/2` or `printf()` when debugging is great, but you need to delete or comment out such calls before submitting your solution. Printing an error to `stdout` when your function is called with invalid arguments is acceptable but not required.

1. **Cracking RSA Keys** (15 points). RSA is a public-key cryptography algorithm. Messages are encrypted using a *public key* and decrypted using a *private key*. The idea is that you can broadcast your public key to the world, and anyone can send you a private message. Only you have the private key, so only you can decrypt the messages that you receive. In RSA the private key consists of two large prime numbers `P1` and `P2` (don't tell anyone these two numbers). The public key is `P1*P2`. RSA relies on the assumption that the time to factor a large composite number is exponential in the number of bits in the number.

   For this problem, we will consider public RSA keys that are too weak to be useful for real-world cryptography. Each of `P1` and `P2` are between 10 million and 20 million. The challenge is:

   > **How many public keys can you crack in under 10 seconds running on `thetis.ugrad.cs.ubc.ca`?**

   The file `hw4_lib.erl` provides a sequential implementation, `crack_keys_seq/3`. You will complete `hw4.erl` and write `crack_keys_par/4`. The file `hw4_lib.erl` also provides functions `test/2` and `test/4` for testing your function `crack_keys_par/4`.

   You need to ensure that your implementation of `crack_keys_par/4` gives the right answer, but you should also try to make it fast. This problem is an example of an embarrassingly parallel problem. See what amazing things you can do. You should also complete the implementation of the function `my_best/0` that returns a tuple of the form `{N_workers, N_RSA_Keys}` where `N_RSA_Keys` is the number of keys you can factor in at most 10 seconds, and `N_workers` in the number of worker processes that you want to have in `W` to achieve this performance. We will use the values from `my_best/0` to confirm your claimed performance.

As a reference point, Mark's solution can factor 33,000 RSA keys where `P1` and `P2` are in [10000000, 20000000] in just under 10 seconds running on `thetis`. Mark's solution is 12 lines of Erlang code. Any solution that cracks 25,000 keys or more in under 10 seconds will get full credit. Solutions which crack 35,000 or more keys in under 10 seconds will get a small bonus, with larger bonuses given for larger numbers of keys cracked. (The extra credit is intended to make this problem *fun* for those who want to see how they can apply the things we've learned about parallel programming and performance analysis and measurement to get a really fast implementation. Don't stress about this, but if you're having fun, then go for it.)

Note: By limiting the execution time to 10 seconds, it should be possible for everyone to complete this assignment without bringing `thetis` to its knees. We recommend that you try to get your timing measurements done *before* the deadline to avoid worries of not being sure you can get all the cores to achieve optimum results.

2. **Getting Started in CUDA** (18 points). Consider the recurrence map $x_i = x_{i-1} + x_{i-2}$ where we assume that x_0 and x_1 are known. With $x_0 = 0$ and $x_1 = 1$ this map generates the famous Fibonacci numbers, while $x_0 = 2$ and $x_1 = 1$ generates the Lucas numbers. In order to get some practice with the CUDA development tools and programming style, in this question we will generate a large number of these sequences with different values of x_0 and x_1 on the GPU, using one thread for each independent sequence. To avoid overflow, we will use a modulo version of the sequence:

$$x_i^{(j)} = (x_{i-1}^{(j)} + x_{i-2}^{(j)}) \mod 2^k \tag{1}$$

where $j = 0, 1, \ldots n-1$ is the index of which sequence we are computing, $i = 0, 1, \ldots, m-1$ is the element within the sequence, and $k \in \{2, 3, \ldots, 32\}$ is small enough that we won't overflow an unsigned 4-byte integer. The parentheses are placed around the superscript to indicate that it is an index not an exponent. If the indexing looks confusing, consider these examples:

$$\begin{gathered}
x_0^{(0)}, x_1^{(0)}, x_2^{(0)}, \ldots, x_{m-1}^{(0)} \text{ is the sequence computed by thread } 0, \\
x_0^{(1)}, x_1^{(1)}, x_2^{(1)}, \ldots, x_{m-1}^{(1)} \text{ is the sequence computed by thread } 1, \\
\vdots \\
x_0^{(n-1)}, x_1^{(n-1)}, x_2^{(n-1)}, \ldots, x_{m-1}^{(n-1)} \text{ is the sequence computed by thread } n-1.
\end{gathered}$$

Remember that we are working in C now, so all indexing is 0-based.

We will consider three versions of computing these sequences:

- "save-last": Given $x_0^{(j)}$ and $x_1^{(j)}$ for all j, compute and return $x_{m-1}^{(j)}$ for all j. Note that the intermediate values $x_i^{(j)}$ for $i = 2, 3, \ldots, m-2$ can be discarded.
- "save-all": Given $x_0^{(j)}$ and $x_1^{(j)}$ for all j, compute and return $x_i^{(j)}$ for all j and all i; in other words, you must save the intermediate elements of each sequence to memory.
- "cum-sum": Given $z_1^{(j)}$ and $y_i^{(j)}$ for all j and $i = 0, 1, \ldots, m-3$, compute and return $z_{m-1}^{(j)}$ for all j which solves the following recurrence

  $$z_i^{(j)} = (z_{i-1}^{(j)} + y_{i-2}^{(j)}) \mod 2^k.$$

  Note that this recurrence is simply calculating the cumulative sum of the values of $y_i^{(j)}$ for $i = 0, 1, \ldots m-3$ starting from an initial sum of $z_1^{(j)}$. However, we will not be doing any kind of fancy reduce tree to solve this problem: Each thread will compute the entire cumulative sum for one value of j using the obvious linear-time loop. Our interest in this cumulative sum arises from the fact that if

  $$z_1^{(j)} = x_1^{(j)} \quad \text{and} \quad y_i^{(j)} = x_i^{(j)} \text{ for all } i = 0, 1, \ldots m-3$$

then the solution $z^{(j)}_{m-1} = x^{(j)}_{m-1}$; in other words, we compute the same result as (1), albeit in a rather round-about manner.

We will implement the "$\mod 2^k$" portion of these calculations by taking the bitwise-and with $2^k - 1$. The two are equivalent in this case. Modern architectures and optimizing compilers should be able to run `%` (C modulo operator) and `&` (C bitwise-and operator) at the same speed, but for some reason modulo is running much slower on the `linXX` machines, so we will use bitwise-and for this assignment.

In the assignment template you will find the following files:

- `recurr-kernel.cu`: The skeleton file where you will write device and host code to implement the three versions of the recurrence described above on the GPU. A number of function stubs have been provided, but you may need to implement additional functions.
- `recurr-main.c`: Driver file which:
  - Implements each of the three versions of the recurrence described above on the CPU. You may find these implementations useful when constructing the GPU implementations.
  - Calls functions from `recurr-kernel.cu` that implement each of the three versions of the recurrence described above on the GPU.
  - Contains code which parses the command line arguments m, n and k, generates the input data for each of the three versions, times the runs, and checks that the final result of all runs are the same.
- `timing418.c`: Helper code to make it easier to collect timings.
- `cuda-helper.cu`: Helper code to decypher various error flags returned by calls to the CUDA libraries.
- `cpu-helper.h`, `cuda-helper.h`, `timing418.h` and `recurr-kernel.h`: Various header files.
- `Makefile`: Ensures that the appropriate flags are used during compilation.

Please remember: **The only file you should modify is** `recurr-kernel.cu`.

Here is what you should do:

(a) (3 points) Download, build and run the template code. At this point the GPU versions will report error(s) because their output arrays are not being computed; however, for the purpose of this part of the question we only want to examine the CPU run times and you can ignore the GPU portion of the output.

Using $k = 24$, try different values of m and n such that the median run times are 0.01 to 1.0 seconds. Briefly describe how the median run times of the three algorithms compare to one another. Based on your understanding of CPU architecture, briefly explain why they might have this relationship. Do not go into a lot of quantitative details—your entire answer should not be more than half a page, and might be much shorter.

(b) (9 points) Modify the file `recurr-kernel.cu` to implement the routines

- `save_last_gpu()`
- `save_all_gpu()`
- `cum_sum_gpu()`
- `save_last_kernel()`

and any other functions you may need (hint: you will probably need additional kernel functions). Your implementations must pass the `check_uint_vectors()` tests included in `recurr_protocol()` in `recurr-main.c`, but you may wish to subject them to additional tests (we certainly will...). For $n > m$ and run times approaching one second, your GPU implementations should be faster than the corresponding CPU implementations; if they are not then you have done something wrong.

(c) (3 points) Build and run your modified code. Using $k = 24$, try some different values of m and n such that the median run times are 0.01 to 1.0 seconds. Briefly describe how the median run times of the GPU implementations compare to one another and to their corresponding CPU implementations. Based on your understanding of GPU architecture, briefly explain why they might have this relationship. Do not go into a lot of quantitative details—your entire answer should not be more than half a page, and might be much shorter.

(d) (3 points) Consider replacing (1) with the recurrence

$$\begin{aligned} x_i^{(j)} &= \left((x_{i-1}^{(j)})^2 + 3x_{i-2}^{(j)} + w_{i-1}^{(j)}\right) \mod 2^k \\ w_i^{(j)} &= (w_{i-1}^{(j)} + 5x_{i-1}^{(j)}) \mod 2^k \end{aligned}$$

where $w_0^{(j)}$ is provided and $k < 16$ so we avoid overflow. Assume that we seek the same outputs for each of the three versions "save-last", "save-all" and "cum-sum"; in particular, there is no need to store or output $w_i^{(j)}$ for any value of i or j once we are done using it. **Without implementing the modified recurrence**, hypothesize what effect this modification would have on the GPU run times of each of the three versions of the recurrence. Your hypotheses should be order of magnitude; for example, "the run time of save-last would decrease by a factor of more than two but less than ten." Based on your understanding of GPU architecture, briefly explain why you drew each hypothesis. Do not go into a lot of quantitative details—your entire answer should not be more than half a page, and might be much shorter.

Supplementary Material: Making the CUDA Template Code Work

We have provided a `Makefile` in the template code, so you should be able to simply call `make` in the appropriate directory. If the build fails mysteriously it is sometimes useful to `make clean` to clear out any stale object files and then `make` again. If the build is successful, you should get an executable binary file whose name is specified by the second line of the `Makefile`.

Executing the binary requires that your `LD_LIBRARY_PATH` environment variable in your shell is set correctly; for example, on the `linXX` machines it should include `/cs/local/lib/pkg/cudatoolkit/lib64`). You can see the value `LD_LIBRARY_PATH` by typing `echo $LD_LIBRARY_PATH` at the prompt. If it does not exist or has no value, you can set it (again on the `linXX` machines assuming that your account is using the default bash shell) with the command:

```
export LD_LIBRARY_PATH=/cs/local/lib/pkg/cudatoolkit/lib64
```

If it has a value which does not include the CUDA path, you can add the CUDA path (again on the `linXX` machines assuming that your account is using the default bash shell) with the command:

```
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/cs/local/lib/pkg/cudatoolkit/lib64
```

You can either run the appropriate command manually every time you log in, or you can add it to your `~/.bashrc` file so it gets run automatically every time you log in.