Toward a Multi-GPU Implementation of the Modular Integer GCD Algorithm

Justin A. Brew & Kenneth Weber
University of Mount Union

Contact Information:
Department of Computer Science
University of Mount Union
1972 Clark Ave., Alliance, OH 44601 USA

Phone: +1 (330) 829 8703
Email: weberk@mountunion.edu

Abstract

The modular integer greatest common divisor (GCD) algorithm [9] holds promise to provide superior performance to sequential algorithms on extremely large input. In order to demonstrate the efficacy of the algorithm, an implementation on a system with multiple Graphics Processing Units (GPUs) is proposed, based on a single-GPU implementation described herein. The implementation's performance is analyzed to predict the size of input needed to demonstrate superior performance when compared to one popular sequential implementation of the integer GCD.

Introduction

Euclid's algorithm to compute the greatest common divisor (GCD) of two integers is one of the oldest algorithms known [7, sect. 4.5.2]. His algorithm describes a process that is inherently sequential, as are most algorithms typically used to compute the GCD, including those currently used by the the GNU Multiprecision Arithmetic Library (GMP) [6]. The modular integer GCD algorithm [9] is unique in that it employs a modular representation for the integer inputs and intermediate results in order to provide a way to parallelize the task. This poster describes an implementation [1] of the modular algorithm on a single NVIDIA graphics processing unit (GPU) [5]. Although its performance is inferior to the GCD operation provided by GMP, there is hope that it can be used as a building block for a multinode implementation that would provide superior performance on very large input values.

Algorithm

The single GPU implementation follows the version of the modular algorithm given in Figure 1. Differences between it and the original include corrections of the following errors in the original [9, Fig. 2]:

- Step MGCD3.2 had a simple typographic error that is corrected in the **forall** starting on line 12.
- Step MGCD4 mistakenly posited that the final result can be computed as each mixed-radix digit and modulus are obtained from the modular representation; lines 23–30 correctly compute all the mixed radix digits and moduli before the recovery of the result begins.

Another difference between the two algorithms stems from the fact that the original algorithm requires a very large number of moduli to guarantee a correct result; in most cases it is possible to use significantly fewer moduli. The algorithm in Figure 1 returns a failure code if the smaller number of moduli chosen is insufficient.

Other Notes:

- a *symmetrical* modular representation is used to represent values in the range $-\lfloor q/2 \rfloor$ to $\lfloor q/2 \rfloor$, instead of 0 to $q-1$.
- inputs are originally in a positional, representation, so conversion to and from modular representation must be done.
- Modular division is simply multiplication by the modular inverse: $u/v \bmod q = u \cdot v^{-1} \bmod q$, where $v \cdot v^{-1} \equiv 1 \mod q$.
- the modular inverse can be computed with an extended greatest common divisor algorithm [7, p. 290].

Input: Positive integers U and V, with $U \geq V$
Output: $\gcd(U, V)$

Constants: $L =$ integer ≥ 2
$\mathcal{M} =$ set of primes in the range $(2^{L-1}, 2^L)$
$C_L = 1.6 - 0.015 \cdot L$

```
1  N_u ← ⌊log_2 U⌋ + 1, N_v ← ⌊log_2 V⌋ + 1
2  N_Q ← ⌈C_L · N_u / log_10 N_u⌉
3  if N_Q > ||M|| then return fail
4  Q ← the set of N_Q largest elements of M
5  forall q ∈ Q do
6      [u_q, v_q] ← [U mod q, V mod q]
7      t_q ← if v_q = 0 then ∞ else u_q/v_q mod q
8  end
9  [p, b] ← [element q of Q for which |t_q| is minimal, t_q]
10 repeat // Reduction loop
11     Q ← Q − {p}
12     forall q ∈ Q do
13         [u_q, v_q] ← [v_q, (u_q − b · v_q)/p mod q]
14         t_q ← if v_q = 0 then ∞ else u_q/v_q mod q
15     end
16     N_Q ← N_Q − 1, [N_u, N_v] ← [N_v, N_v − L + ⌊lg b⌋]
17     if N_Q (L − 2) ≤ N_u then // Can't recover G
18         return fail
19     [p, b] ← [element q of Q for which |t_q| is minimal, t_q]
20 until b = ∞
21 k ← 1, G ← 0
22 [p_1, g_1] ← [element q of Q with priority to u_q ≠ 0, u_q]
23 repeat // Recover mixed-radix representation
24     Q ← Q − {p_k}
25     forall q ∈ Q do u_q ← (u_q − g_k)/p_k mod q
26     k ← k + 1
27     [p_k, g_k] ← [element q of Q with priority to u_q ≠ 0, u_q]
28 until g_k = 0
29 for i = k − 1 downto 1 do G ← g_i + p_i G
30 return |G| // Return standard representation
```

Figure 1: Modular algorithm, as implemented

Implementation Notes

- The implementation uses moduli in the range $2^{31} < q < 2^{32}$.
- A technique of Cavagnino and Werbrouck [2] allows us to compute $x \bmod q$ via 64-bit multiplication rather than 64-bit division.
- Our use of Cavagnino and Werbrouck's technique restricts the number of usable moduli to 68 million out of 98 million 32-bit primes.
- Synchronization among the thread blocks—needed on lines 9, 19, 22 and 27—is done by either cooperative groups [5, Appendix C] or a custom-built barrier employing spin-waiting.
- Cooperative groups are stabler and scalable, but the barrier is faster on Pascal GPUs and necessary for older GPUs.
- Global minimum is performed by combining two levels of reduction operations in each thread block (using warp shuffle [5, Appendix B.15]) with synchronization among the thread blocks.

Results

The single-GPU implementation was executed on several different GPUs, given randomly generated inputs of various sizes. Times for each execution are plotted in Figure 2 below, against the time needed to execute the GCD operation from GMP on a reference CPU—an Intel Xeon E5-2620.

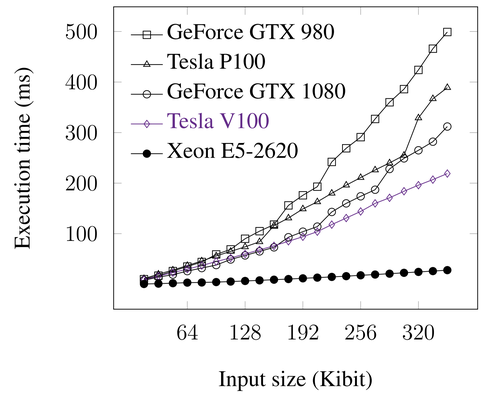

Figure 2: Modular GCD times vs GMP GCD time

Figure 3 highlights the behavior of the ModGCD algorithm on the Tesla V100. In addition, a least-squares linear fit to the first ten times given for the V100 is plotted, which provides us with a means to predict the behaviour of the algorithm, given enough execution resources (i.e., cores) to allow it to exhibit its linear behavior in terms of execution time. Admittedly, this ignores possible non-linear communication behavior.

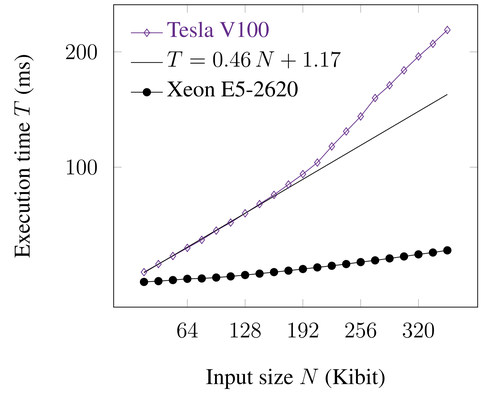

Figure 3: Modular GCD behavior on Tesla V100

Conclusions

- A multinode implementation of the algorithm would become superior to the GMP GCD implementation at inputs of around 250 million bits (modular = 112 sec vs. GMP = 121 sec).
- The 68 million 32-bit moduli available would sustain GCD computation on inputs up to 529 million bits.

Future Work

We intend to build and test a multinode implementation on the Owens supercomputer at the Ohio State University [4]. In order to demonstrate efficacy, it will be necessary to employ a larger system, such as the Summit supercomputer at the Oak Ridge National Laboratory [8].

References

[1] Justin Brew and Kenneth Weber. 2018. ModGCD-OneGPU. (2018). Retrieved May, 2018 from https://github.com/MountUnionComputerScienceDepartment/ModGCD-OneGPU/releases/tag/v1.2-alpha

[2] D. Cavagnino and A. E. Werbrouck. 2008. Efficient Algorithms for Integer Division by Constants Using Multiplication. *Comput. J.* 51, 4 (2008), 470–480.

[3] Ohio Supercomputer Center. 1987. Ohio Supercomputer Center. http://osc.edu/ark:/19495/f5s1ph73. (1987).

[4] Ohio Supercomputer Center. 2016. Owens supercomputer. (2016). http://osc.edu/ark:/19495/hpc6h5b1

[5] CUDA C Programming Guide 2018. CUDA C Programming Guide. (2018). Retrieved April 4, 2018 from http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf version 9.1.85.

[6] Torbjörn Granlund. 2016. *GNU MP: The GNU Multiple Precision Arithmetic Library* (6.1.2 ed.). Free Software Foundation.

[7] Donald E. Knuth. 1997. *The Art of Computer Programming* (3rd ed.). Vol. 2: Seminumerical Algorithms. Addison-Wesley, Reading, Massachusetts.

[8] Summit ORNL 2018. System User Guide: Summit. (2018). Retrieved March 6, 2018 from https://www.olcf.ornl.gov/for-users/system-user-guides/summit/

[9] Kenneth Weber, Vilmar Trevisan, and Luiz Felipe Martins. 2005. A Modular Integer GCD Algorithm. *Journal of Algorithms* 54, 2 (February 2005), 152–167.

Acknowledgments

The authors thank the Ohio Supercomputer Center [3] for access to a Tesla P100 GPU on the Owens system, as well as Anthony Rizzo for his contributions to initial software development, NVIDIA Corporation for the donation of a Tesla C2070 GPU, and Seneca Data Distributors (a subsidiary of Arrow Electronics) for the donation of computer time for testing of early versions of the project.