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

KENNETH WEBER
University of Mount Union
Department of Computer Science
Alliance, OH, USA
weberk@mountunion.edu

JUSTIN A. BREW
University of Mount Union
Alumnus, Department of Computer Science
Alliance, OH, USA
jbrew5662@gmail.com

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.

CCS CONCEPTS

• **Theory of computation → Massively parallel algorithms**;

KEYWORDS

GPU, Integer GCD

ACM Reference Format:
KENNETH WEBER and JUSTIN A. BREW. 2018. Toward a Multi-GPU Implementation of the Modular Integer GCD Algorithm: Extended Abstract. In *Proceedings of 47th International Conference on Parallel Processing (ICPP 2018).* ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 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. What follows describes an implementation [1] of the modular algorithm on a single NVIDIA graphics processing unit (GPU) [5] that could be used as a foundation for a multinode implementation providing superior performance on very large input values.

2 ALGORITHM OVERVIEW

The implementation of the modular algorithm described here uses the variant of the original given in Figure 1. It *estimates* the number of moduli that will be needed and checks to see whether there will be enough moduli to finish the computation at each iteration of the reduction loop. It also incorporates corrections to two errors in Steps MGCD3.2 and MGCD4 of the original algorithm [9].

```
Input: Positive integers $U$ and $V$, with $U \ge V$
Output: $\gcd(U, V)$
Constants: $L$ = integer $\ge 2$
           $\mathcal{M}$ = set of primes in the range $(2^{L-1}, 2^L)$
           $C_L = 1.6 - 0.015 \cdot L$
$N_u \leftarrow \lfloor \log_2 U \rfloor + 1$, $N_v \leftarrow \lfloor \log_2 V \rfloor + 1$
$N_Q \leftarrow \lceil C_L \cdot N_u / \log_{10} N_u \rceil$
if $N_Q > ||\mathcal{M}||$ then return fail
$Q \leftarrow$ the set of $N_Q$ largest elements of $\mathcal{M}$
forall $q \in Q$ do
    $[u_q, v_q] \leftarrow [U \bmod q, V \bmod q]$
    $t_q \leftarrow$ if $v_q = 0$ then $\infty$ else $u_q / v_q \bmod q$
end
$[p, b] \leftarrow$ [element $q$ of $Q$ for which $|t_q|$ is minimal, $t_q$]
repeat // Reduction loop
    $Q \leftarrow Q - \{p\}$
    forall $q \in Q$ do
        $[u_q, v_q] \leftarrow [v_q, (u_q - b \cdot v_q)/p \bmod q]$
        $t_q \leftarrow$ if $v_q = 0$ then $\infty$ else $u_q / v_q \bmod q$
    end
    $N_Q \leftarrow N_Q - 1$, $[N_u, N_v] \leftarrow [N_v, N_v - L + \lfloor \lg b \rfloor]$
    if $N_Q (L - 2) \le N_u$ then // Can't recover $G$
        return fail
    $[p, b] \leftarrow$ [element $q$ of $Q$ for which $|t_q|$ is minimal, $t_q$]
until $b = \infty$
$k \leftarrow 1$, $G \leftarrow 0$
$[p_1, g_1] \leftarrow$ [element $q$ of $Q$ with priority to $u_q \neq 0$, $u_q$]
repeat // Recover mixed-radix representation
    $Q \leftarrow Q - \{p_k\}$
    forall $q \in Q$ do $u_q \leftarrow (u_q - g_k)/p_k \bmod q$
    $k \leftarrow k + 1$
    $[p_k, g_k] \leftarrow$ [element $q$ of $Q$ with priority to $u_q \neq 0$, $u_q$]
until $g_k = 0$
for $i = k - 1$ downto $1$ do $G \leftarrow g_i + p_i G$
return $|G|$ // Return standard representation
```

Figure 1: Modular algorithm, as implemented

ICPP 2018, August 13-16, 2018, Eugene, OR USA
© 2018 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

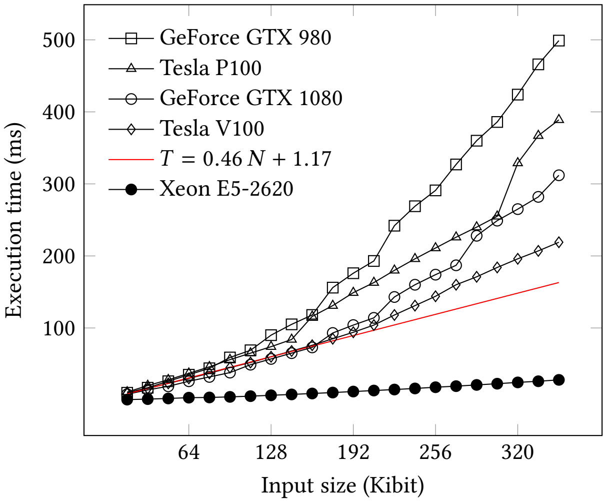

Figure 2: Modular GCD times vs GMP GCD times

3 IMPLEMENTATION DETAILS

In this section we provide a few key implementation details.

Although division is slow, NVIDIA GPUs support native 32-bit integer operations [5]. Therefore, the implementation uses $L = 32$ to get as much benefit from hardware support as possible. A technique provided by Cavagnino and Werbrouck in [2] allows us to compute the remainder when dividing a 64-bit value by a 32-bit modulus via 64-bit multiplication, rather than 64-bit division. This is significantly faster than the code generated by the C++ compiler for the 64-bit remainder operation, but restricts us to the use of only 68 million of the 98 million 32-bit primes. By requiring somewhat more work per modulus at runtime [2, sect. 2.4], all 32-bit primes could be used.

The global minimum needed on lines 9 and 19 of Figure 1 is performed by combining two levels of reduction operations using warp shuffle [5, Appendix B.15] in each thread block, together with synchronization among the blocks; global “any” (lines 22 and 27) combines the `__ballot_sync` function [5, Appendix B.13] with block synchronization. It was originally necessary to implement our own global barrier to synchronize thread blocks. NVIDIA recently introduced Cooperative Groups [5, Appendix C], which provide a reliable means to synchronize among thread blocks. Cooperative Groups are not available on some NVIDIA GPUs, and appear to be slower than the global barrier on others, so results are *not* reported for the version of the implementation that uses Cooperative Groups.

4 RESULTS

Figure 2 displays the performance of the implementation on several NVIDIA GPUs. Ten pairs of inputs were randomly generated for each size reported, and the average execution time for the ten pairs is plotted. Times recorded were actual physical times needed to compute the values; care was taken to reduce, as much as possible, any system activity that may be included in the measurements. A least-squares fit of the first 10 execution times for the Tesla V100 GPU, the most powerful device (and having the most recent architecture) of those included in the graph, is also provided.

5 CONCLUSION

Given enough processing elements, the modular GCD algorithm should exhibit linear behavior [9]. The GCD algorithms used by GMP for large input are essentially $O(N^{1+\epsilon}\log N)$ for a fairly large value of ϵ [6, section 15.3.3], so—given enough processing elements—a multinode implementation of the modular GCD algorithm should be faster than the GMP implementation for very large input.

It can be seen from Figure 2 that the execution times exhibit linear behavior up to the point at which the device becomes saturated by the number of threads it must support, which appears at input sizes of 160 Kibit for the Tesla V100. Using the linear least-squares fit displayed in the figure, we predict that the modular algorithm will be faster than the GMP implementation for inputs of over 250 million bits, at which value GMP GCD took 121 seconds for one pair of 250 million bit inputs, and our extrapolation projects a running time of around 112 seconds on a hypothetical multinode system with enough GPUs of the same type as the projection is based on, and with fast enough communication between GPUs. With 68 million usable 32-bit moduli, a multi-GPU implementation should be able to handle input sizes of up to 529 million bits, based on the formula for N_Q from line 2 of Figure 1. Although this projection encourages further investigation, only an actual multinode implementation of the modular algorithm will allow a definitive assessment of its efficacy.

6 FUTURE WORK

We plan to develop a multi-GPU implementation on the Owens supercomputer at the Ohio State University [4] with the goal of demonstrating efficacy on a larger system, such as the Summit supercomputer at the Oak Ridge National Laboratory [8].

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.

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.