15-712 Project Report: The Performance of Batch-Parallel Modular Exponentiation on CPU and GPU

Sam Westrick Benjamin Berg Laxman Dhulipala

ACM Reference Format:
Sam Westrick Benjamin Berg Laxman Dhulipala. 2019. 15-712 Project Report: The Performance of Batch-Parallel Modular Exponentiation on CPU and GPU. In *Proceedings of ACM Conference (SPAA '19).* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Hardware today is increasingly specialized for particular tasks. Perhaps the most prominent example is the GPU, which first emerged in the late 90s with increased demand for fast video processing. Although GPUs are specialized for graphics, this does not mean that their use is limited to graphics applications. There has been a remarkable push in the past decade for using GPUs to solve a variety of "embarrassingly parallel" tasks, including scientific computing, machine learning, cryptography, etc.

GPU proponents claim massive speedups (e.g. multiple orders of magnitude improvement) in comparison to traditional approaches on a standard CPU technology. However, there are potential issues with this claim [9]. There is a fundamental difference in design between high-performance libraries for the CPU and GPU: the GPU is assumed parallel, whereas the CPU is often assumed single-threaded. A prime example is the GNU Multiple Precision Arithmetic Library (GMP) [1], which implements a variety of fast bignum primitives for CPUs, however all of GMP's primitives are designed to solve a single problem instance at a time. In contrast, Nvidia's XMP library [2] implements fast multiple precision arithmetic on *batches* of problem instances. This begs the question: are there any performance gains to be had by switching to a batch parallel model on the CPU? More broadly: how much of the benefit of GPU is due to the assumption of a batch parallel model?

In this project, we aim to gain some understanding of these questions by considering a particular problem: fast modular exponentiation on the CPU and GPU. The problem is simple (given b, e, and m, compute $b^e \mod m$) but fundamental to a variety of cryptographic applications.

1.1 Overview and Contributions

The structure of the rest of this paper is as follows:

- In Section 2, we explain the state-of-the-art algorithm for modular exponentiation, and highlight where the expensive calls in this algorithm lie.
- In Section 3.1, we describe the hardware specific capabilities exploited by XMP and our experience trying to port this code to a CPU based implementation.
- In Section 3.2, we describe the process of extending the GMP implementation of the Montgomery algorithm with a batch-parallel variant, taking advantage of low-level vectorization to achieve performance improvement.
- In Section 4, we evaluate our GMP-based implementation, and discuss directions for future work.

2 MONTGOMERY MULTIPLICATION

Montgomery multiplication is an optimized algorithm for computing the product of two positive integers, a, b modulo a modulus m. The method is useful for computing $a^n \mod n$, a ubiquitous computation in cryptosystems like RSA. The motivation for the Montgomery algorithm is twofold. First, we compute many repeated multiplications (a repeatedly multiplied by itself) modulo a fixed number n. Left unchecked, the bit-complexity of the operand will become huge and finding the modulus of this huge operand modulo n will be very costly. The idea in the Montgomery algorithm is to repeatedly apply a *reduction* operation, which takes the result of multiplying two numbers a and b in $[0, n)$ and efficiently reduce the produce modulo n.

At a high level, the technique replaces a modulus operation modulo m, which is a costly division operation with a division by another integer r s.t. $gcd(m, r) = 1$. In practice (and in these notes) m will always be odd, and therefore we can choose r to be the first power of two larger than m so that the division and modulus just become bitshift and masking operations. Note that this requirement of m being odd is satisfied in practice since we are usually computing modulo a large prime.

Classical modular multiplication. In classical modular multiplication, to compute the quantity $ab \mod m$ we first compute the product ab which may require 2 machine words if a, b are both w bits each, and then perform a division on this double-word product. There are two costly steps: the first is switching to operations on double-words, and the second is division by an arbitrary m which necessitates repeatedly subtracting m until the value falls below m.

2.1 The Montgomery Algorithm, modulo Reduction

We now present the Montgomery algorithm, while treating the key step—the reduction algorithm—as a black box. Note

SPAA '19, June 2019, Phoenix, AZ, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn