Chris Kolegraff
CSC 314
Program 1

Big Integer Factorial Program

Introduction

The goal of this project is to use my knowledge of ARM Assembly to speed up a program that calculates large factorial numbers. The program uses a big integer struct to hold values that are much bigger than any standard 32 or 64-bit number could hold. A big integer is defined as a struct with an unsigned integer array which acts as a single large number. I have been given a file that contains the source code, the big integer header file, a timing script and a make file. The original code is in C, but Dr. Pyeatt has rewritten the *bigint_adc* function in Assembly already. The task is to speed up the implementation of the rest of the algorithm using Assembly and any other performance enhancements we can think of.

The timing script is used to generate a total runtime of the program. It will sum together the times of 10 runs of the program, and return the total time taken and the speedup of the program compared to the original code. Below is a list of functions that have been modified or created in Assembly.

Functions Modified / Created in ASM

bigint_mul_uint	bigint_shift_left_chunk
bigint_alloc	bigint_free
bigint_from_int	bigint_smallmod
accumulate_noTrim	qdiv64by10

Table 1: List of functions created or rewritten in ASM

Timings and Speedups

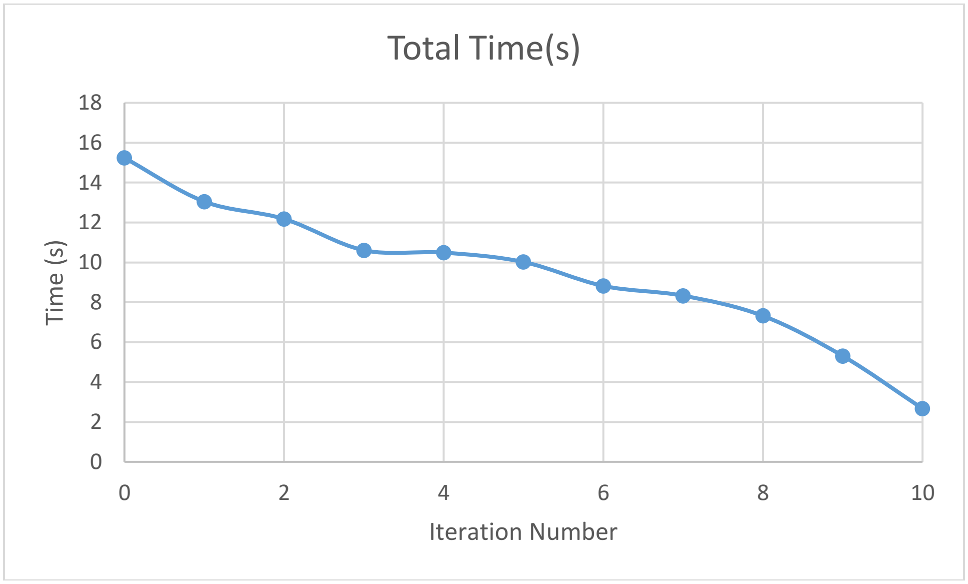
Multiple functions have been rewritten and optimized for speed in Assembly. Each function modified will have its own section on the specifics of what was changed in terms of the coding structure and of the algorithm involved in performing that task. The table below will describe the iteration number, the function worked on, that iteration's total time, and the speedup of the program relative to the original code. Below the table is a graph which will illustrate the incremental speed increases throughout the iterations.

To generate the timings, I ran './report_time.sh' to get the time it took to run the program 10 times, and the amount of speedup that was generated.

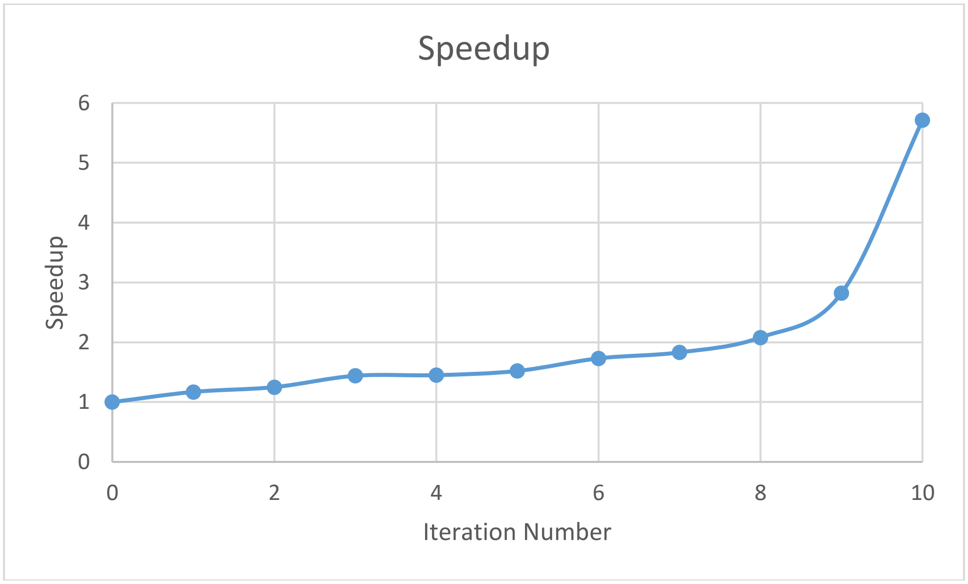
Iteration	Function Name	Total Time(s)	Speedup
0	Original C Code (with bigint_adc)	15.24	1
1	bigint_mul_uint	13.05	1.17
2	bigint_shift_left_chunk	12.18	1.25
3	bigint_from_int	10.61	1.44
4	bigint_alloc / bigint_free	10.49	1.45

5	bigint_smallmod (Divide)	10.02	1.52
6	bigint_smallmod (Divide by 10)	8.82	1.73
7	bigint_smallmod (Improvements)	8.33	1.83
8	bigint_mul (Accumulate)	7.32	2.08
9	bigint_mul_uint (Accumulate)	5.41	2.82
10	bigint_mul (Dynamic Memory)	2.67	5.71

Table 2: Table containing Iteration number, function, total time, and speedup



Graphic 1: A chart describing the total time vs. iteration number



Graphic 2: A chart describing the speedup vs. iteration number

Iteration Details

Iteration 0 – Original C Code (with bigint_adc)
This is the original code that was given to work on. The only function that was written in Assembly was the *bigint_adc* function. This iteration is the one that will be compared to for the other iterations of this program.

Iteration 1 – bigint_mul_uint
The first thing that I had to change was the unsigned multiply between a bigint and an unsigned integer. There were some problems with the original algorithm. One such problem was that the bigint variable ‘tmp1’ was being dynamically allocated and freed inside the multiplication loop. That meant that it was allocating and deallocating the same bigint of size 3 repeatedly. Dynamic allocation of memory is already incredibly slow, so one strategy for speeding up a program is to limit the amount of memory being allocated and freed in an algorithm. I will be revisiting this concept in later iterations.

I have also decided to make the big integer struct an unsigned big integer struct because the factorial program only requires multiplying numbers between 2 and 150. This means that I have removed the code that checks for negative numbers, and that I am assuming no negative numbers are passed in to the function. I have also removed the code to check if the unsigned integer passed in, ‘r’, is greater than the CHUNKMASK define because we know for certain that we are working on a 32-bit word system, and those checks involved redefining an integer as an 8 or 16-bit long chunk.

Iteration 2 – bigint_shift_left_chunk
This was the next function that was taking the biggest percentage of execution time besides the *__udivdi3* function, which I did not understand where that function was coming from at the time. The biggest change I made was changing the number of array indices. Instead of having one integer to access both arrays, I have two integers. This way, I can avoid doing unnecessary adds and subtracts to get the correct array access. One was for the tmp->blks array, and the other was for the l->blks array. This also made the Assembly code easier to read and understand, and made it faster than the original algorithm as well.

Iteration 3 – bigint_from_int
The divide function was still the function with the biggest percentage of execution time, but I was still researching where it was coming from. The next biggest percentage came from *bigint_from_*int. The original C code that was written did not assume that the program was to be run only on a 32-bit system. It has code that uses the current chunk size and the size of an integer to determine how many blocks to allocate in the bigint. That algorithm used some dividing as well, which slowed the program down. Since we know that this program is to be run on a 32-bit system, I have instead written the function to allocate a bigint with a size of 1, and then I set the passed in value into the bigint, and return it.

Iteration 4 – bigint_alloc / bigint_free
Both functions were in the top 6 functions with the biggest percentages of runtime, so I thought I should optimize these functions for speed, however, there was not much speedup when I

converted these functions to Assembly. My thinking was that these functions were being called numerous times, so I figured writing them in Assembly would limit the number of clock cycles spent on these functions.

Iteration 5 – bigint_smallmod (Divide)
I did not write my own *bigint_smallmod* function in C and translate it into Assembly, instead, I wrote my own *bigint_smallmod* function in Assembly from that start. I chose to do this because I wanted to avoid using the divide functions. When I used *grpof* I saw that the function *__udivdi3* was taking the most time to execute by far. I wanted to avoid these slower divides so I ended up using Dr. Pyeatt's divide functions on the website. These worked well for this iteration, but at this point I was not satisfied with the amount of speedup I received from doing this.

Iteration 6 – bigint_smallmod (Divide by 10)
For my next iteration, I decided to implement the divide function in another way for this function. I knew that the only time *bigint_smallmod* was called was to determine the next digit in the decimal equivalent to the bigint, which meant dividing by 10 and getting the remainder. I decided to implement the multiply reciprocal method to dividing, and added a special case that if the input was '10', then it would execute the faster algorithm. I created the function *qdiv64by10* which gave me a good amount of speedup, however, I was still not satisfied with the overall algorithm in the *smallmod* function.

Iteration 7 – bigint_smallmod (Improvements)
For this iteration, I tried to rewrite the algorithms used in both the *bigint_smallmod* and *qdiv64by10* functions to utilize less of the stack, and to perform quicker. I also added some conditional code in the *qdiv64by10* code to use different multiplication algorithms depending on whether a high or low word in the input was empty, ultimately reducing the time it would take to perform the multiply overall. I also changed the final multiplication algorithm to get the remainder with bit shifts and adds, however, I did not see a large improvement in speedup from that change.

Iteration 8 – bigint_mul (Accumulate)
Using *grpof* I found that the *bigint_adc* function was the thing that was taking the most amount of time in the program. I decided to create my own adding algorithm that was specifically meant to be used for the two multiplication algorithms. I did this because *bigint_adc* does some dynamic memory allocation, and it also can trim the end of bigints of words that are all 0's. This significantly slows down the program. I wrote a function called *accumulate_noTrim* which expects two bigint inputs, and it performs an add and stores the result back into one of the inputs. It does this with no memory allocation, and it does not check to make sure the result is a reasonable size.

Iteration 9 – bigint_mul_uint (Accumulate)
The same thing was happening in this function that was happening in Iteration 8. After looking at *gprof* I found that the *bigint_adc* was still the function taking the largest portion of time to run. I then modified by *bigint_mul_uint* function to account for the new accumulation function I wrote, and inserted it into the algorithm. After using *gprof* again I found that *bigint_adc* had dropped significantly in the amount of overall time that was spent on it.

Iteration 10 – bigint_mul (Dynamic Memory)
The biggest issue that I saw with the multiplication algorithm was that all the functions used in the multiplication loop had some form of dynamic memory allocation in it. This significantly slowed down the multiply loop. I modified the functions used by *bigint_mul*: *bigint_mul_uint*, *bigint_shift_left_chunk*, and *accumulate_noTrim*. These functions no longer dynamically allocate memory, instead, they take in big integers as inputs, and the output is stored in one of the passed in bigints.

At the start of the multiply algorithm, I allocate space for four bigints, and initialize the ones that need to be initialized. I then start the multiplication loop. Due to using the *accumulate_noTrim* function, the result may have many blocks that represent leading 0's. After the second to last iteration of the loop, the loop will break and execute the same multiplication algorithm found in the loop, but instead of using the accumulation function, it will use the *bigint_adc* function to trim the result back to its correct size.

Conclusion
After rewriting multiple functions, and creating a couple of my own, I increased the speed of the entire program by a factor of 5.71. Both the dividing and multiplying functions took large amounts of time for varying reasons. For the *bigint_smallmod* function, the problem was using a divide strategy that was incredibly slow. For the *bigint_mul* and *bigint_mul_uint* functions, the issue was having too much dynamic memory allocation in the multiplication loops, resulting in needlessly allocating large chunks of memory. More functions could be rewritten in Assembly to further speed up the program, but I believe the changes that would result in the largest amounts of speed increase have already been made to the program.