Arbitrary Precision Integers on the Cell Processor

New Mexico

Supercomputing Challenge

Final Report

April 6, 2010

Team #36

Desert Academy

Team Members

Megan Belzner

Matt Rohr

Bjorn Swenson

Teacher / Project Mentor

Thomas Christie

Table Of Contents

0 Executive Summary

The Cell processor is used in the Playstation 3 gaming console and the Roadrunner supercomputer at Los Alamos National Labs. It is a microprocessor specially designed to do calculations quickly with high data throughput. It has eight cores to process data and one core to manage the data flow, and provides many advantages, such as efficient calculation with the vector data type which allows multiple operations to be carried out in a single CPU cycle. In addition, all of the cores can be used at the same time and can communicate between each other.

Despite all this, there has never been any efficient use of the Cell for arbitrary precision calculations. Our goal in this project was to make an arbitrary precision integer library that utilized the special capabilities of the Cell to allow for extremely fast calculations on any size integer.

We programmed our library from scratch on a Playstation 3 using C. We created a new data type, called a vecthor. Each vecthor is an array of vectors that can hold up to [any multiple of] 1024 digits. We then proceeded to create functions for the standard operations commonly used in number theory and cryptography, namely addition, subtraction, multiplication, modding, and taking powers.

Throughout the creation of our library, we paid special attention to efficiency in memory usage and speed. We utilized many Cell-specific functions, and constantly tested our functions for speed. We collected data for multiple number lengths, and compared our results to the leading arbitrary precision library, GMP.

Our intention was to create a library optimized for the Cell processor that could be implemented separately and simultaneously on all of the cores and on multiple networked PS3s. We intended it to be used primarily for number theory applications such as prime finding as well as encryption and decryption.

However, even with meticulous code optimizations and the power of the Cell processor, our library failed to meet the mark. When compared to GMP, the results were dismal. GMP outperformed us by several orders of magnitude in most cases, even for relatively simple operations. We concluded that the power of the Cell processor is very hard to extract and manipulate. In general, the difficulty of programming and managing memory is more trouble than it is worth when dealing with foreign data types and machine instructions.

1 Introduction

In the year 2001, Sony, Toshiba and IBM (collectively called "STI") began work on the Cell Broadband Engine (CBE). The Cell is a microprocessor with a heterogeneous design: 8 cores processes data, and one core controls job execution and manages data flow. It was built around IBM's PowerPC technology, and is used in both Sony's Playstation 3 gaming system and IBM Supercomputers, including the Roadrunner at Los Alamos National Labs. All applications, from modeling physical simulations to multimedia, required that the processor have superior vector and floating point performance.

In its current design, the Cell has 9 cores - one PPU (Power Processing Unit) and eight SPUs (Synergistic Processing Unit), all of which run at 3.2GHz. The PPU is an ordinary PowerPC processor and serves to organize data flow and job execution between the other cores. The SPUs are specially designed processors theoretically capable of 25.6 GFLOPs each, and can perform operations on 128-bit data types called vectors. IBM and third parties have developed libraries on the Cell tailored for graphics and modeling, such as arbitrary precision floating point arithmetic and fast matrix multiplication. However, currently no arbitrary precision integer library exists which utilizes the Cell's unique capabilities, meaning the Cell is not being used for number theory or cryptography.

1.1 Problem

Our goal with this project was to create an arbitrary precision integer library for the Cell processor using the C programming language. The Cell processor on the Playstation 3 has a PPU (main processor) and 6 usable SPUs, since one is dedicated to the PS3's operating system and one is disabled by default. As mentioned above, each SPU can do basic arithmetic operations on 128-bit data types, or 4 integers at once. This means that, depending on the parallelizability of the algorithms, we thought we might be able to increase the speed of arithmetic operations on a single core up by up to 4x. If we could make each SPU operate at maximum capacity, each Cell could do 6 to 24 times as many arbitrary-precision operations as a single-core 3.2GHz processor in a given amount of time. Combined with the fact that MPI can be used to make multiple PS3s work on a problem in tandem, we thought that several networked PS3s could operate as a formidable machine for number-theoretical operations like prime finding and encoding/decoding information.

2 Implementation

We chose to work on the Cell because of the calculating speed of the 6 usable individual SPUs, the availability the Cell in the form of the Playstation 3, and the SIMD (Single Instruction Multiple Data) capabilities of the Cell Processor. The SPU's 128-bit vector data type can be divided in various ways, such as 4 integers, 8 shorts, 2 longs, etc. Included in the Cell's libraries provided by IBM are functions called intrinsics which map directly into assembly language instructions. One such intrinsic function allows multiple numbers to be added together in a single computation like so:

Vector 1:	{230597,	17345,	5198357,	3567}
	+	+	+	+
Vector 2:	{3298673,	839764,	235,	46082}
	=	=	=	=
Result:	{3529720,	857109,	5198592,	49649}

This effectively means that 4 integers can be added simultaneously.

We created a new data type to take advantage of the speed of the Cell's intrinsic functions, which are SPU functions that map directly onto assembly language instructions. Our library of functions is called "thor," so we named our data type "vecthor." Each vecthor consists of an array of [some multiple of] 32 vectors, plus one more that contains the size of the vecthor. Each vector contains 4 integers of maximum 8 digits each, and the number is broken into 8-digit groups and inserted in the vectors. Of course, a signed integer can have a value up to 2^{31} = 2 147 483 648, but we chose 99 999 99 as our "BIGGEST_INT" so that we could add several vectors together without the results overflowing the 2^{31} value.

In a vecthor, the first vector contains the metadata: the first integer (as well as the last one) is how many vectors are registered in memory as being part of the vecthor (not including the size vector and always a multiple of 32), the second integer contains the “virtual” size – how many vectors are actually used to contain the number, and the third integer contains the sign (0 for positive, -1 for negative).

For example the number 34673892005592058499301472901653829271067 38 in vecthor form is:

{32, 2, 0, 32} *30 vectors w/ only 0s* {0, 0, 346, 73892005} {59205849, 93014729, 01653829, 27106738}

We also decided to code in C for raw speed and usability in C++ code.

2.1 Single vs. Multi-core Operations

When we started our project we wanted to make use of the Cell's communication abilities by using Direct Memory Access (DMA) to allow all six SPU's to work on a single operation at the same time. However this approach would have far more difficult to implement than expected and the constant data transfer required would have hurt the speed. So with expert advice from DeLesley Hutchins we decided to focus on creating operations for single SPUs, and planned to have the PPUs dole out jobs to each SPU.

Once we decided to make an operation run on a single SPU, we considered using the GNU Multi-Precision integer library (GMP) and modifying it to utilize the vector functions of the SPU. Unfortunately, we could not get GMP to compile for the SPU, presumably to the SPU's limited built-in functionality. We therefore proceeded to create a SPU-optimized library from scratch.

2.2 Data Type Development

In our original implementation, we dynamically sized the vecthors so that they always had just enough vectors to contain the number (plus the meta vector). However, this meant that the vecthor had to be resized whenever the number overflowed the current vecthor, and a lot of time was wasted on memory allocation. For example, if advanced carrying was necessary in addition with the old system, for a function c=func(a,b), first c had to be put in a temporary vecthor, then c had to be re-initialized to be one vector larger. Then the process had to be repeated as soon as the new vector was filled, which in the large calculations this is designed for was quite often. One call of the C functions calloc and malloc takes about 340 times as much time as an intrinsic add or subtract operation. With dynamic resizing, over 50% of the time for a single arbitrary-precision subtraction operation was used to allocate memory, and almost 50% of addition was allocating memory.

With a mostly fixed-size vector, overflow happens far less often – and the vector is simply doubled in size, which means that it’s unlikely to overflow again any time soon and much less time and resources are wasted on resizing the vecthors. A 2048 bit prime number key for RSA encryption is roughly 600 decimal digits. We chose a 32 vector vecthor as it holds 1024 digits, so for these smaller prime numbers this means we won't have to resize the vecthors. Vethors now double in size when the numbers exceed 1024 digits, 2048 digits, etc and halve with getting smaller.

2.3 Limited Success

Because of the nature of the Cell, our intial goal was to have the PPU manage jobs for the SPUs, while the SPUs performed their jobs extremely quickly and reported back to the PPU. Unfortunately, we quickly realized that programming at this low-level on a foreign processor architecture was unrealistic based on time constraints, and essentially beyond the scope of our project. As a compromise, we decided to run programs on the SPU alone. There are a few advantages to this:

-Writing code in C using vectors was much more easier to understand than working around the complexity of the DMA (Direct Memory Access) Transfers, individual mailbox routing to and from SPUs, and context/thread creation and management for each SPU.

-Using vectors on an SPU would give us a reasonable understanding on whether or not the Cell's philosophy and technology is really a breakthrough. Therefore, we could write and run code relatively quickly while still being able to draw a conclusion on the Cell as a whole.

-The SPUs are known to be the fastest part of the Cell. A comparison of the new, exotic SPUs to a normal intel processor would be much more beneficial than one of the power PC brand PPU to an intel processor.

No arbitrary precision integer library currently exists for the Cell, and if the processor's power was truly revolutionary, we may have seen dramatic increases in calculations with prime numbers, for example. We gave our library the name 'thor', and decided to answer the question everyone is asking: Are the speed benefits from the Cell *really* worth the hassle of programming on a completely unexplored, foreign processor architecture? Cleve Moler assured us that they are not, and as we discuss below, we have come to the same conclusion (at least for this application).

2.4 Function List

Below is a list of all the functions we created, with a description of how they work and how they are used. The full code is provided in appendix A.

typedef vector signed int* __attribute__ ((aligned(4096))) vecthor;

Summary: Defines the size of a vecthor.

vecthor init_vecthor(void);

Summary: Initializes a 4096-bit vecthor and returns a pointer to that vecthor

Description: Our function for initializing vecthors simply allocates 32 vectors, each vector consisting of 128 bits (4096 bits for a standard vecthor). This is done using the calloc function, which allocates and initializes a block of memory. The metadata is also appropriately initialized, and finally a pointer to this newly created vecthor is returned. Initially, we passed the desired size of the vecthor as the first argument. However, after we switched to a fixed-sized datatype implementation, this was no longer needed.

void vecthor_clear(vecthor a);

Summary: Zeros out a vecthor, setting all values to zero and updating metadata accordingly.

Description: This function loops through each vector in the vecthor, and inserts the scalar value of zero into each element of these vectors. The metadata is also adjusted for the altered values.

void vecthor_copy(vecthor a, vecthor b);

Summary: Copies all data from *a* into *b*.

Description: To copy data from one vecthor to another, we simply loop through the first vecthor, placing all vectors in the source vecthor into the destination vecthor. This memory copying is admittedly slow. However, if one was to simply set the two vecthors equal to each other (e.g. *a=b;*), this would only set the pointers equal to each other, therefore causing all further operations performed on *a* also to be performed on *b*.

*void str_to_vecthor(vecthor a, char *str);*

Summary: Sets *a* equal to numeric data found in *str*.

Description: str_to_vecthor works as follows: First, loop through the vectors in *a* from right to left. For each iteration in the loop, convert the next 8 characters in *str* into an *int*, then place this in the relevant element in the vector.

void vecthor_write_virtual_size(vecthor a);

Summary: Determines the size of the number in the 4096-bit vecthor spaceand adjusts the meta-data vecthor accordingly. Used primarily in other library functions

Description: In order to determine the variable virtual size of the data inside a vecthor (rather than the actual size), we look at each vector in the vecthor from left to right. If, at any time during this loop, the given vector's contents are *not* equal to 0, then the loop is terminated and the size is updated in the metadata. This loop must go from left to right, otherwise a series of 32 zeros or more could possibly be interpreted as the end of the data in the vecthor (therefore corrupting the virtual size).

int vecthor_virtual_size(vecthor a);

Summary: Returns the virtual size of the vecthor, i.e. the number of vectors containing data. Used primarily to optimize loops in library functions.

Description: Literally, this function returns the second element in the metadata vector. This variable can be used to optimize nearly every core arithmetic operation in the library. The reason for this is that if a vecthor consists of a 32 vector-long chain, but only 1 or 2 of these vectors hold actual data, then it is useless to loop through this empty data (to perform addition or subtraction, for instance). Using the *vecthor_virtual_size()* function, we could easily tell where a vecthor's data started and ended, and thus could loop through only the data that mattered when optimizing our functions.

int vecthor_actual_size(vecthor a);

Summary: Returns the actual size of a vecthor, i.e. the number of 128-bit vectors allocated to the vecthor.

Description: As opposed to *vecthor_virtual_size()* (see above function), the actual size of a vecthor is the amount of vectors allocated to the particular vecthor. It follows that in order to determine the number of bits a vecthor occupies in memory, this conversion can be used:
memory_usage(a)=128**vecthor_actual_size(a);*
as a vector takes up 128 bits.

int vecthor_sign(vecthor a);
Summary: Returns the sign of a vector (1 if positive, -1 if negative, 0 otherwise).
Description: This function returns the third element of the metadata vector.

void print_vector(vector signed int a);
Summary: Prints a single vector (not a vecthor).
Description: This function is used in debugging, to examine the individual elements of a vector. Normally, *print_vecthor()* or *print_vecthor_debug()* is desired.

void print_vecthor(vecthor a);
Summary: Prints an entire vecthor to screen in an easy-to-read fashion.
Description: *print_vecthor()*, after determining the relevant size bounds of the vecthor, takes all data vectors and displays them on the screen. Extra precaution is taken in ensuring that the padding is correct (e.g. eight zeros will not be shortened to one, and zeros on the left will not be displayed).

void print_vecthor_debug(vecthor a);
Summary: Prints an entire vecthor to screen, including meta-data and zeros
Description: This function quite literally prints all data (relevant or not) that the vecthor occupies in memory onto the screen. This is used in testing functions, and is generally excessively difficult to read.

void vecthor_lightning_add(vecthor c, vecthor a, vecthor b);

Summary: As fast as the lightning of thor! Computes the sum of *a* and *b*, storing the result in *c*. This function does not carry for you, so it should only be used when it is certain that no overflow will occur.
Description: This function uses the assembly language intrinsic *spu_add()* to perform simple addition on two vecthors. It is best used when incrementing a larger vecthor by a smaller amount, or for adding small values.

void vecthor_add(vecthor c, vecthor a, vecthor b);
Summary: Computes the sum of *a* and *b*, storing the result in *c*.
Description: Addition is performed using the same *spu_add()* assembly language routine, however, carrying (the heart of addition) is performed afterwards to ensure that the data is lined up correctly.

void vecthor_carry(vecthor c);
Description: Ensures that each element is 99999999 or below. Properly formats the vecthor. Use after any operation that increases the size of entries.
Summary: This function is a carry function for use in addition. It utilizes the fact that we only use 8 digits per vector entry while an int can go up to 9. It loops through each entry of each vector, dividing the entry by 100,000,000 (our biggest allowed int plus 1) to obtain the carry and calculating the entry mod 100,000,000 to obtain the "mod". The carry is added to the next entry to the left, and the mod replaces what was previously in the entry.

void vecthor_carry_ext(vecthor c, int biggest);
Summary: Carries, but with the option of manually setting the biggest-int size
Description: Instead of treating the biggest int as 99999999, the biggest int is specified by the user.

void vecthor_sub(vecthor c, vecthor a, vecthor b);
Summary: Computes the difference of *b* and *a*, storing the result in *c*.

Description: Subtraction is first performed normally (e.g. simply subtracting everything in *b* from *a*, not worrying about negative values or borrowing). Next, we look at all negative values. For each negative value, we:

-Take the absolute value of this number, making it positive.

-Subtract this positive value from our biggest int + 1 (100,000,000).

-Subtract one from the value to the left of it in the vecthor.

void vecthor_mul(vecthor c, vecthor a, vecthor b);

Summary: Computes the product of *a* and *b*, storing the result in *c* using the slow but straightforward naive multiplication. The product must be less than 1024 digits.

Description: Naive multiplication is very similar to multiplication by hand. We first loop through all the vectors in *b* and multiply each of these vectors by every vector in *a*. The result is stored in an *unsigned long long*. The result is split into two parts (the first part is the division by the biggest int, the second part is the long long mod our biggest int), and added to the appropriate 'slot' in the vecthor. Periodically, the result vecthor must be carried to prevent overflow.

void vecthor_pow(vecthor c, vecthor a, vecthor bb);

Summary: Uses successive squaring to compute *a* raised to the *b* power, storing the result in *c*. *b* must be less than 99999999.

Description: Successive (sometimes called repeated) squaring involves looking at the binary representation of *b* (the exponent). If the LSB is one, then our result vecthor is multiplied by our base. Regardless of *b*'s binary representation, our base is square with every iteration. At the end of each iteration, our exponent is shifted right. The loop terminates when the exponent is equal to zero.

void vecthor_mod(vecthor c, vecthor a, vecthor b);

Summary: Computes *a* mod *b*, storing the result in *c*.

Description: Uses an algorithm that shifts *b* to the left until its binary representation is one bit less in length than *a*. Then, *b* is subtracted from a until *a*<*b*. This is repeated until *a* is less than the original *b*.

void vecthor_powm(vecthor c, vecthor a, vecthor bb, vecthor m);

Summary: Computes *a* raised to *b* (mod *m*) and stores the result in *c* using successive squaring.

Description: See *vecthor_pow()*. The algorithm is identical, except *a* is squared and subsequently modded by *m* with each iteration.

void vecthor_is_prime(vecthor c, vecthor a);

Summary: Uses Fermat's Little Theorem to test the primality of *a*. *a* must be less than 99999999. Splats the result vecthor with 1's if *a* is prime, 0's otherwise.

Description: Fermat's Little Theorem uses a small prime number as a base (2, 3, 5, 7), and raises this to *a-1*, modding the result by *a*. If the result is not 1, then *a* is definitely not prime. If the result is one, then we repeat the process with a different base. The more bases used, the surer one can be that *a* is prime.

int vecthor_comp(vecthor a, vecthor b);

Summary: Compares the size of a and b. Returns 1 if a>b, 0 if a==b, and -1 if a<b.

Description: This function loops through *a* and *b,* using assembly language intrinsics to determine if one vector is greater than another. If one is found to be greater than another, then the loop terminates.

int vecthor_cnt_lz(vecthor a);

Summary: Counts leading binary zeros in a vecthor.

Description: This function counts the binary leading zeros in a vecthor by using a loop coupled with the *spu_cntlz* function.

void vecthor_lshift(vecthor c, vecthor a, int times);

Summary: Multiplies vecthor *a* by 2 *times* number of times using a left-shift intrinsic function and carrying, storing the result in *c*.

Description: This function calls *spu_slqw()* and carries when appropriate to prevent overflow or data discrepancies.

int vecthor_bin_diff(vecthor a, vecthor b);

Summary: Used in modding to determine how many times *b* would have to double to have the same size binary representation as *a*.

Description: This function simply subtracts the values of *vecthor_cnt_lz()*.

int vecthor_pos_extract(vecthor a, int pos);

Summary: Extracts the number in position *pos* from vecthor *a* where position is measured from right to left, (ie. the rightmost integer has pos=0 and the leftmost would have pos=32, since a standard size vecthor holds 32 integers). Used exclusively as an auxiliary function in multiplication.

void vecthor_pos_mul_add(vecthor a, int pos, unsigned long long x);

Summary: Takes *x*, which represents two ints multiplied, split it into parts greater than and less than 99999999 and add it to the values to existing values in *a*. Used exclusively as an auxiliary function in multiplication.

int vecthor_find_max_pos(vecthor a);

Summary: Returns the "size" of a, counting from left to right. Max is 32*4=1024. Used exclusively as an auxiliary function in multiplication.

void start_timer(void);

Summary: Starts SPU timer, measured in ticks. One tick is approximately equal to 40 clock cycles.

int read_timer(void);

Summary: Reads from SPU timer, returning the number of "ticks" since the timer started.

double ticks_to_ms(int x);

Summary: Converts x ticks to milliseconds, returning the number of ms for a given number of ticks.

void init_aux(void);

Summary: Should be called at the beginning of *main*. Initializes a number of auxiliary vecthors used to extend the functionality of certain operations.

Description: This function simply calls *init_vecthor()* on a number of global auxiliary vecthors. These are used when memory overwriting may occur (e.g. calling *vecthor_add(a,a,b)* may cause memory discrepancies).

int add_elements(vector signed int a);

Summary: Adds all of the elements in a given vector, returning the result.

int intpow(int x, int pow);

Summary: Computes *x* raised to the *pow* power. Used in *str_to_vecthor*.

int abs(int x);

Summary: Returns the absolute value of an integer using bit logic to avoid branching.

2.5 Software Used

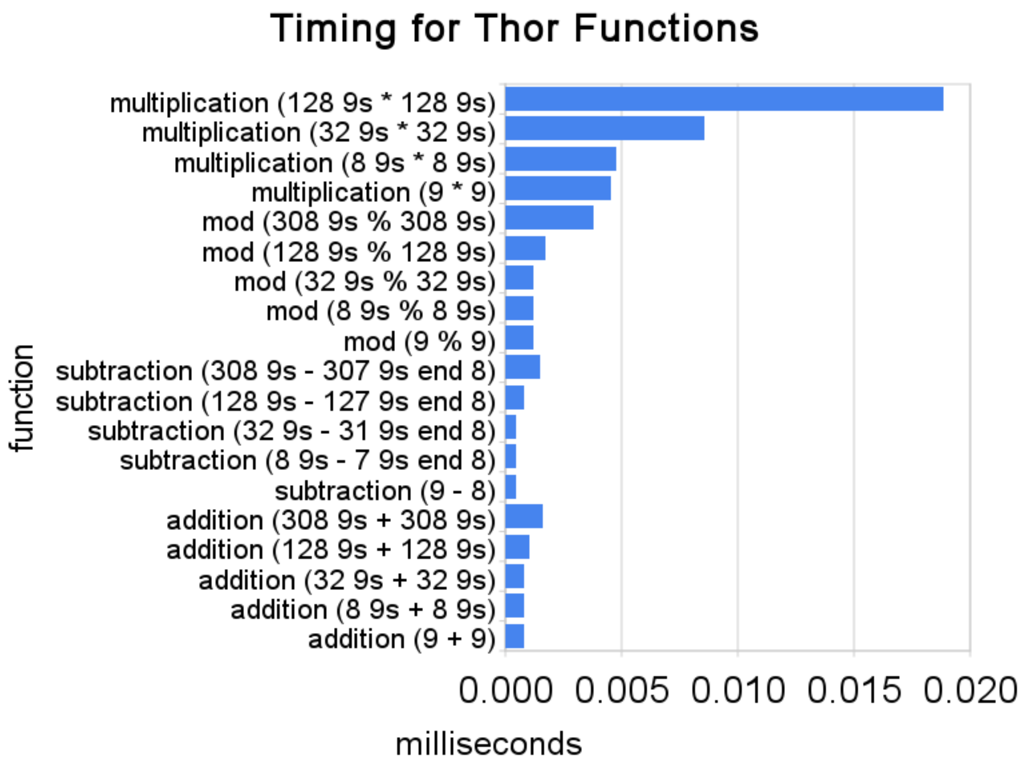
To enable code writing and compilation on the Playstation 3 systems, we installed Fedora or Yellow Dog Linux distributions on each system. Recent firmware updates have disabled the option to install a third-party OS, though Sony has disabled and re-enabled this capability in the past. To compile C code on the Cell's PPU and SPU cores, we used Cell-specific gcc builds created by IBM named spu-gcc and ppu-gcc. These are available for free on the IBM website as part of the “Software Development Kit for Multicore Acceleration Version 3.0,”

2.6 Literature Used

For instructional material, we used both books and IBM manuals for development on the Cell. For general information on programming in C, we used Aitken’s *Teach Yourself C in 21 Days* and Kernighan and Ritchie’s *The C Programming Language*. For Cell specific information, we used *Programming the Cell Processor*, by Matthew Scarpino. This was our single best resource, without which we could not have produced as much Cell-specific code as we did. The IBM manuals we used were all part of the Cell SDK. Citations for these sources are provided at the end of this document.

3 Results

Below is a chart of the time taken for a single run-through of our primary functions. We tested each function with different numbers, once with 1 digit (e.g. 9+9), once with a full entry in a vector (99 999 999 + 99 999 999), once with a full vector (32 9s + 32 9s, etc), once with 4 full vectors, and one test with 308 digits which is roughly the length of a 1024-bit number, and therefore a component of a 2048-bit RSA key.

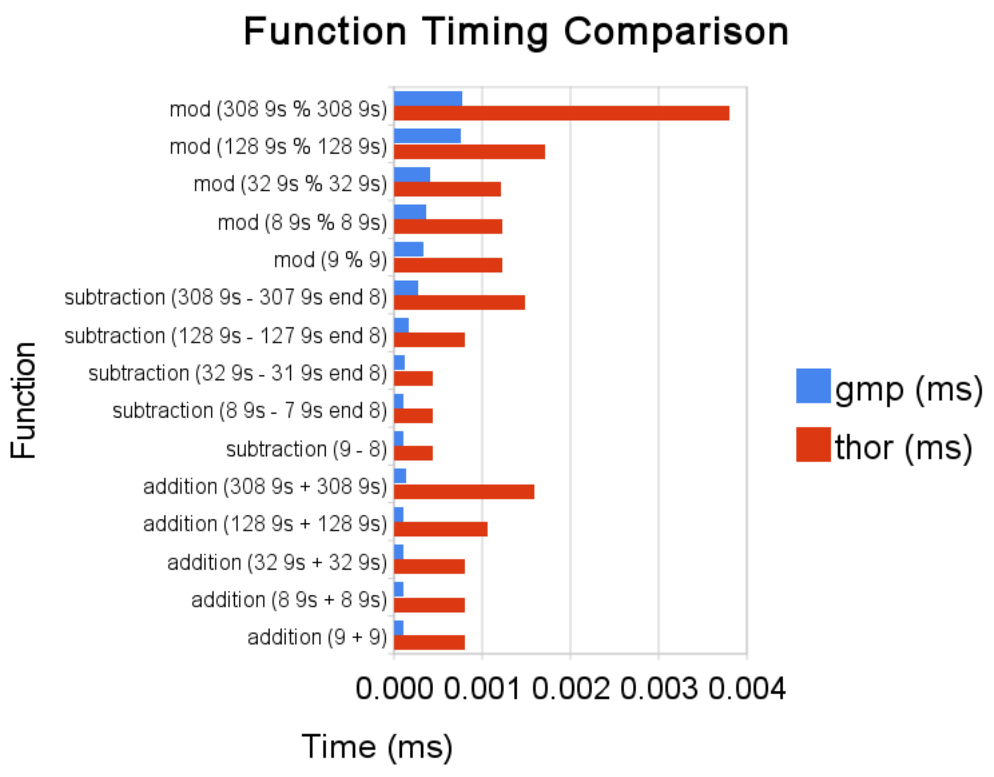


We measured the timing in SPU ticks, which is equal to 40 processor cycles. To get more comprehensible view of the time taken, we translated the values in ticks into milliseconds by dividing the values by 80,000 (3.2 billion cycles per second, 3.2 million cycles per millisecond / 40 cycles per tick = 80,000 ticks per millisecond).

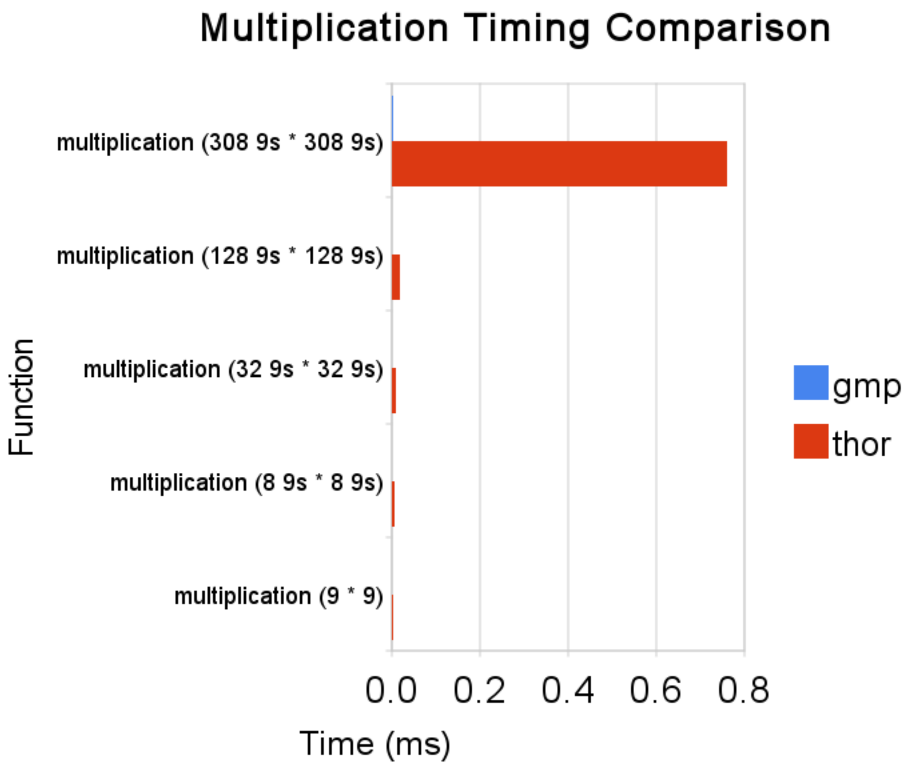
3.1 Comparison to GMP

The real test came when we compared the timing of our functions to the industry standard of arbitrary precision arithmetic, the GMP (GNU multiple precision) integer library. For the GMP data below, we used the PPU, which runs at the same 3.2GHz clockspeed. Converting PPU and SPU results to milliseconds allowed us to compare them directly. Of course, different processor architecture is being used in each case, but as GMP will not compile on the SPU it is as close as we could get to a true comparison.

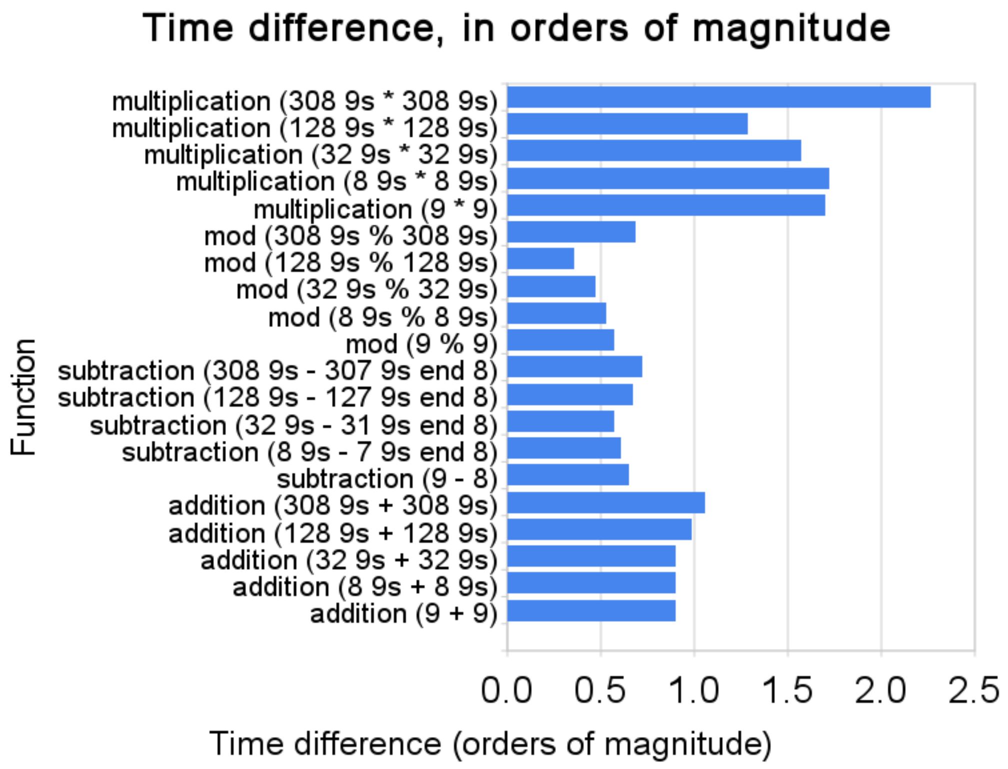
These graphs show that GMP's functions far outperform thor's functions.



The timing below is correct. While our adding and subtracting functions likely use similar algorithms to those used in GMP, our multiplication is naive multiplication (the type you do by hand), and is therefore very slow compared to GMP's more sophisticated multiplication algorithms.

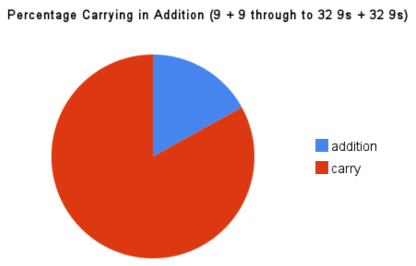


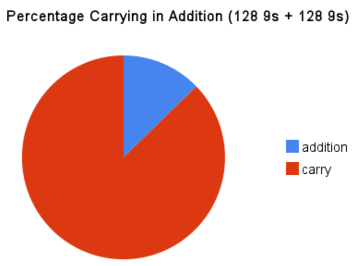
While GMP obviously outperforms thor for reasons discussed below, most functions are within an order of magnitude difference. Multiplication is noticeably different, but as mentioned above, GMP utilizes Fast Furier Transform for multiplication, so we expected ours to be much slower. The comparison is like apples to oranges until we develop FFT multiplication for the Cell.

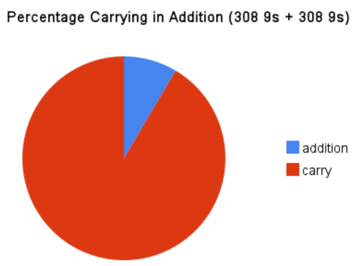


This graph shows the log of the ratio between thor's functions and GMP's functions. It displays the time difference between comparable functions in terms of order of magnitude (i.e. 10^2) to allow for better visibility of the data.

The following charts show the percentage of total time in an addition operation used in adding and carrying. The percentage of time used for carrying is roughly 83%, 87% and 91% respectively. For addition operations smaller than 9 digits, carrying takes up roughly 83% of the total time as well.







4 Analysis

While we thought the Cell's novel structure would prove to be faster than a normal computer, the results proved just the opposite. The fact that SPUs are not good at branching coupled with the fact that datatypes can not be directly manipulated by C lead our library to be much slower than we thought it would be. When compared to the GNU Multiple Precision library (GMP), we found that in several cases we were many orders of magnitude behind. While the Cell processor may sound good on paper, vectors may be more trouble than they're worth, at least for integer arithmetic.

4.1 Parallelizability / Comparison to GMP

The Cell processor is praised for its ability to be parallelized and optimized for working with four integers at once (vectors). However, when it comes to integer math, algorithms are only partially parallelizable. For instance, when carrying after performing an addition operation, every significant bit relies explicitly on the bits less significant than it, and the algorithm must operate from right to left. Thus, even with 6 or 7 SPUs, addition can not be optimized for multiple processors. This does not lie inherently in the Cell processor, but is rather a weakness of positional notation in general. Modulus and multiplication do not parallelize well either: our implementation of modulus is reliant on the results of repeated subtraction and shifting, and multiplication still involves the implementation of a carry method. Because of this, arbitrary precision libraries tend to be inherently slow, and some algorithms *must* be linear in fashion. Even if the core of the operation can be parallelized, the carry must still be done linearly. This greatly stunts the ability to optimize algorithms such as these.

Another reason GMP outperformed thor was because of the difficulty of memory access on the Cell. On a modern CPU, one can simply execute *a[2]+=10;*. The processor is already adept at dealing with arrays of numbers. However, on the Cell, that piece of code would resemble this: **a=spu_insert(spu_extract(*a,2)+10,*a,2);*. This is actually three operations, each of which takes about 3 CPU cycles. The Cell must copy an integer from a place in memory, add to it, and insert it again into memory. This lengthy bit of code leads to more compiled instructions, less efficient memory management, and ultimately less efficient code. This leads us to conclude that when it comes to raw computing power in relation to small individual operations, the Cell is definitely not the weapon of choice.

We expected our addition and subtractions to be most comparable to those utilized by GMP, because they are the most obviously and simply parallelizable. Indeed, the actual adding or subtracting is remarkably easy. However, carrying involves several operations for each element of each component vector:

Step 1: extract an element from a vecthor, mod it by BIGGEST_INT + 1
Step 2: add the result to the next entry over
Step 3: divide the extracted element by BIGGEST_INT + 1
Step 4: Insert the result of that operation back into its original place

In C, it looks like this, where *(c+i) is the entry being processed

```
temp=spu_extract(*(c+i),j);
*(c+i)=spu_insert(spu_extract(*(c+i),j-1)+temp/(biggest+1),*(c+i),j-1);
*(c+i)=spu_insert(temp%(biggest+1),*(c+i),j);
```

If the Cell had built-in intrinsic functions designed for integer modding and integer division, this process would be much faster. Unfortunately, it does not. As it is, this sequence of oeprations must be done for every single entry of a vecthor. We suspect that this carrying operation is what makes our adding algorithm slower than GMP's, not the adding *per se*. As it was, results indicated that 80-90% of the CPU cycles in an adding operation were used for carrying. If carrying were able to see the same 4x increase that pure addition does, an operation that now takes 100 time units would take roughly 28 time units. This would make our operations take 3x as long as they do using GMP, rather than 10x.

With that said, a well-parallelizable, higher level algorithm may in fact see a speed benefit from the Cell architecture. If work was divided up in an efficient way and doled out to the SPUs intelligently, each one could perform their small bit of the problem sequentially (the part of the problem would have to be large enough as to not suffer from the above speed penalties). For example, when generating a large prime for use in an RSA key, each SPU could be asked to look at a different set of numbers, meaning the overall result would be found 6x as fast as using one SPU. However, in this particular example, since GMP cannot be compiled on the SPUs and thor is not fast enough to be worth using, a standard multi-core processor using GMP is still a better choice. Given the successful application of the Cell in physical simulations and multimedia, it seems the Cell is great for floating point computations that are not strictly sequential. However, the Cell does not show appreciable value under the constraints of integer arithmetic. The above considerations also suggest that using highly parallelized architectures such as GPUs or upcoming GP-GPUs for doing integer math would probably not be worthwhile.

5 Conclusion

Through exploring the capabilities and limitations of the new Cell processor architecture, we have learned not only of some specific drawbacks to using the Cell, but some drawbacks of parallelized computing in general. For example, while vectors may be able to hold four integers and operate on them simultaneously, the strenuous task of reading from and writing to these vectors when operating on them is slow. Also, when performing numeric operations, because of the nature of arithmetic algorithms, some pieces of data rely on other pieces. This leads to a large stunting in the amount of data that can be parallelized at a low level, and, therefore, a smaller amount of potential in multicore programming to be realized.

The Cell's SPU cores reputedly lack efficient branching capabilities, which also greatly hurt us. Many arbitrary precision libraries examine the input and select an algorithm based on length of the operands. On the Cell however, determining which algorithm to use on specific input data would be extremely time consuming, and this limited our speed greatly. Keep in mind that we ran into branching problems on algorithms as low of a level as adding. A higher level algorithm might experience even more severe impediments, depending on its design.

On a larger scale, we can safely conclude that when examining a program with the intent of parallelizing it, one should not necessarily attempt to distribute the low-level number crunching loops, but rather examine the algorithm from a top down perspective and optimize it in this fashion. However, the textbook way of optimizing a program for multiple cores typically says the opposite. One is usually told to look for bottle-neck loops in the program and divide these up between available processors. From our extensive experience on the Cell, we can say that this is in fact *not* the way to go about it. The main reasons for this are the Cell processor's sheer incompetence when it comes to completing low level sequential integer operations smoothly. Because these low level operations require a large number of functions to accomplish relatively simple tasks, the Cell benefits from a top-down optimization approach. Most of our optimization mistakes were made in assuming that the best way to optimize was to tweak functions at as low of a level as possible.

5.1 Most Significant Original Achievement

Our most significant achievement is taking several algorithms we take for granted when using high level languages (addition, subtracting, multiplication, string and memory manipulation), and examining them closely at the lowest possible level. It has taught us a great deal about what it was like to implement the algorithms, as well as specific details as to how compilers, bits in memory, memory allocation, assembly language, threading, processor architecture, and intense line-by-line optimization works. The final product, spu_thor.h, may or may not be used extensively by the Cell programming community (or lack thereof). However, through producing the library we have learned more than ever before about the nature of computers and algorithms, and it will continue to influence the way we meticulously build and optimize our programs.

6 Resources

Aitken, Peter G., Bradley Jones, and Peter G. Aitken. *Sams Teach Yourself C in 21 Days*. Indianapolis, Ind.: SAMS, 2000. Print.

"DeveloperWorks : Cell Broadband Engine Resource Center." *IBM - United States*. Web. 07 Apr. 2010. <http://www.ibm.com/developerworks/power/Cell/>.

Kernighan, Brian W., and Dennis M. Ritchie. *The C Programming Language*. New Delhi: Prentice-Hall of India, 1999. Print.

Scarpino, Matthew. *Programming the Cell Processor: for Games, Graphics, and Computation*. Upper Saddle River, NJ: Prentice Hall, 2009. Print.

We would also like to give special thanks to the Los Alamos National Lab Foundation for providing the funding used to purchase the equipment used in our project. Without their help, this project would have been difficult to impossible. Thank you!

Appendix A – Code

```
/*****THOR Arbitrary Precision Integer Library for the Cell Processor
*****
****** Version 0.1a
****** Currently designed to operate on one SPU
******
****** The basic data type is a "vecthor," which is an array of
vectors, each containing 4 32-bit integers.
****** Each vecthor represents an arbitrary precision integer
******
****** To use, place this file in the same directory as your SPU source
code and #include "spu_thor.h"
****** Note: You must also have the IBM Cell Development Library
Installed
******
******/

#include <stdio.h>
#include <spu_intrinsics.h>
#include <string.h>
#include <spu_mfcio.h>

#define BIGGEST_INT 99999999
#define BIGGEST_SHORT 9999
#define INTS_IN_VEC 4
#define SHORTS_IN_VEC 8
#define VECS_IN_VECTHOR 32
#define NUM_AUX 6
#define ADD_AUX 0
#define SUB_AUX 1
#define MUL_AUX 2
#define POW_AUX 3
#define MOD_AUX 5

typedef vector signed int* __attribute__
((aligned(128*VECS_IN_VECTHOR))) vecthor;

vecthor aux[NUM_AUX];

vecthor           init_vecthor(void);
// Initializes a 4096-bit vecthor and returns a pointer to that vecthor
// use: a=init_vecthor()

void             vecthor_clear(vecthor a);
//Zeros out a vecthor and changes meta-data accordingly
//use: vecthor_clear(a)

void             vecthor_copy(vecthor a, vecthor b);
//Copies from a to b
//use: vecthor_copy(a,b)

void             str_to_vecthor(vecthor a, char *str);
//Creates a vector with the same value as a given string
```

```
//Must initialize first using init_vecthor
//use: str_to_vecthor(a,"123456")

void            vecthor_write_virtual_size(vecthor a);
//Determines the size of the number in the 4096-bit vecthor space
//and adjusts the meta-data vecthor accordingly.
//Used primarily in other library functions

int                 vecthor_virtual_size(vecthor a);
// Returns the virtual size of the vecthor, i.e. the number of vectors
containing data
// Used primarily to optimize loops in library functions

int                 vecthor_actual_size(vecthor a);
//Returns the actual size of a vecthor,
//i.e. the number of 128-bit vectors allocated to the vecthor
//use: a_size=vecthor_actual_size(a);

int                 vecthor_sign(vecthor a);
//Returns the sign of a vector
//1 if positive
//-1 if negative
//0 if zero
//use; a_sign = vecthor_sign(a)

void            print_vector(vector signed int a);
//Prints a single vector (not a vecthor)
//use: print_vechor(a);

void            print_vecthor(vecthor a);
//Prints an entire vecthor to screen
//use: print_vecthor(a);

void            print_vecthor_debug(vecthor a);
//Prints an entire vecthor to screen, including meta-data and zeros
//use: print_vecthor_debug(a)

void            vecthor_lightning_add(vecthor c, vecthor a, vecthor
b);
//Adding without carrying - use with care, when you know you won't have
to carry!
//adds a and b, stores the result in c
//use: vecthor_lightening_add(c,a,b)

void            vecthor_add(vecthor c, vecthor a, vecthor b);
//Adding a and b, storing the result in c
//use: vecthor_add(c,a,b)

void            vecthor_carry(vecthor c);
//Goes through the vecthor, ensuring that each int is 99999999 or below
//Makes the vecthor properly formatted. Use after any operation that
increases
//the size of entries
//use: vecthor_carry(c);

void            vecthor_carry_ext(vecthor c, int biggest);
//Carries, but with the option of manually setting the biggest-int size
```

```
//use: vecthor_carry_ext(c)

void             vecthor_sub(vecthor c, vecthor a, vecthor b);
//Subtracts b from a, stores in c
//use: vecthor_sub(c,a,b)

void             vecthor_mul(vecthor c, vecthor a, vecthor b);
//Multiplies a and b, storing the result in c
//Multiplies using naive multiplication (slow, but all we have right
now)
//use: vecthor_mul(c,a,b)
//At this point, the result must be smaller than 1024 digits for this
to work properly

void             vecthor_pow(vecthor c, vecthor a, vecthor bb);
//Uses successive squaring to take a^b and store it in c
//Only works when b is less than 99999999
//use: vecthor_pow(c,a,b)

void             vecthor_mod(vecthor c, vecthor a, vecthor b);
//Takes a%b and stores in c
//Uses an algorithm that shifts b until its binary representation is
one bit less in
//length than a, subtracts, and repeats until a is less than b
//use: vecthor_mod(c,a,b)
//STILL WEIRD - fails with some large values

void             vecthor_powm(vecthor c, vecthor a, vecthor bb, vecthor
m);
//Takes (a^b)%m and stores the result in c
//Uses successive squaring
//Only works when bb is less than 99999999
//use: vecthor_powm(c,a,b,m)

void             vecthor_is_prime(vecthor c, vecthor a);
//Uses Fermat's Little Theorem to test the primality of a.
//Only works when a is less than 99999999
//Sets c to {1,1,1,1} if prime, {0,0,0,0} otherwise
//use: vecthor_is_prime(c,a)

int                 vecthor_comp(vecthor a, vecthor b);
//Compares size of a and b. Returns:
// 1 if a > b
// 0 if a = b
// -1 if a < b
//use: result=vecthor_comp(a,b)

int                 vecthor_cnt_lz(vecthor a);
//Counts leading binary zeros in a vecthor
//Note:An empty vector will have 128 zeros, so
//an empty vecthor will have 32*128 = 4096 leading zeros
//An easy way to tell if a vecthor equals zero
//use: lz = vecthor_cnt_lz(a)

void             vecthor_lshift(vecthor c, vecthor a, int times);
//Multiplies a vector a by 2 "times" number of times using
//an left-shift intrinsic function and carrying, storing the result
```

```
//in c
//use: vecthor_lshift(c,a,5) //multiplies a by 32, for example

int                 vecthor_bin_diff(vecthor a, vecthor b);
//Used in modding to determine how many times b would have to double to
//have the same size binary representatino as a

int                 vecthor_pos_extract(vecthor a, int pos);
//Used in multiplication to extract the number in position "pos" from
vecthor a
//where position is measured from right to left, ie. the rightmost
integer has pos=0
//and the leftmost would have pos=32, since a standard size vecthor
holds 32 integers

void             vecthor_pos_mul_add(vecthor a, int pos, unsigned long
long x);
//Used in multiplication to take x, which represents two ints
multiplied, split it into parts greater
//than and less to 99999999 and add it to the values already in a

int                 vecthor_find_max_pos(vecthor a);
//Returns the "size" of a, counting from left to right. Max is
32*4=1024
//Used in multiplication

void                start_timer(void);
//Starts SPU timer, measured in ticks, each of which represents about
40 clock cycles
//use: start_timer()

int                read_timer(void);
//reads from SPU timer, returning the number of "ticks" since the timer
started
//use: t1 = read_timer();

double             ticks_to_ms(int x);
//Converts x ticks to milliseconds, returning the number of ms for a
given number of ticks
//use: t2 = ticks_to_ms(t1)

void             init_aux(void);
//Initiates an axillary vector, to be used as a holding vector in some
operations
//use: int_aux()

int                add_elements(vector signed int a);
//Adds all of the elements in a given vector, returning the result
//use: sum = add_elements(a)

int                 intpow(int x, int pow);
//Takes the power of an integer, because C doesn't have a power
function built in for some reason
//Doesn't use successive squaring, so use sparingly.  This is used in
str_to_vecthor.
//use: answer = intpow(x,y)
```

```
int                 abs(int x);
//Returns the absolute value of an integer
//use: x_abs = abs(x)


vecthor init_vecthor(void){
    vecthor a;
    a=(vecthor)calloc(VECS_IN_VECTHOR+1,sizeof(vector signed int));
    *a=(vector signed int){VECS_IN_VECTHOR,1,0,32};
    return a;
}


void vecthor_clear(vecthor a){
    int i,lim=vecthor_actual_size(a);
    for(i=1; i<=lim; i++){
        *(a+i)=(vector signed int)spu_splats(0);
    }
    *a=spu_insert(1,*a,1);
}


void vecthor_copy(vecthor a, vecthor b){
    int i,sizeoffset=vecthor_actual_size(b)-vecthor_actual_size(a);
    for(i=vecthor_actual_size(a)-vecthor_virtual_size(a)+1;
i<=vecthor_actual_size(a); i++){
        *(b+i+sizeoffset)=*(a+i);
    }
}


void str_to_vecthor(vecthor a, char *str){
    int i,j,k,m=0,tmp,len=strlen(str),numvex=1+((len-1)/32);

    *a=(vector signed int){VECS_IN_VECTHOR,numvex,0,32};
    for(i=vecthor_actual_size(a); i>vecthor_actual_size(a)-numvex; i--
){
        *(a+i)=spu_splats(0);
        for(j=3; j>=0; j--){
            tmp=0;
            for(k=7; k>=0; k--){
                tmp+=(str[len-m-1]-'0')*intpow(10,7-k);
                m++;
                if(m>=len){*(a+i)=spu_insert(tmp,*(a+i),j); return;}
            }
            *(a+i)=spu_insert(tmp,*(a+i),j);
        }
    }
}


void vecthor_write_virtual_size(vecthor a){
    int i=1,size=spu_extract(*a,0);
    while(add_elements(*(a+i))==0 && size){i++; size--;}
    *a=spu_insert(size,*a,1);
}
```

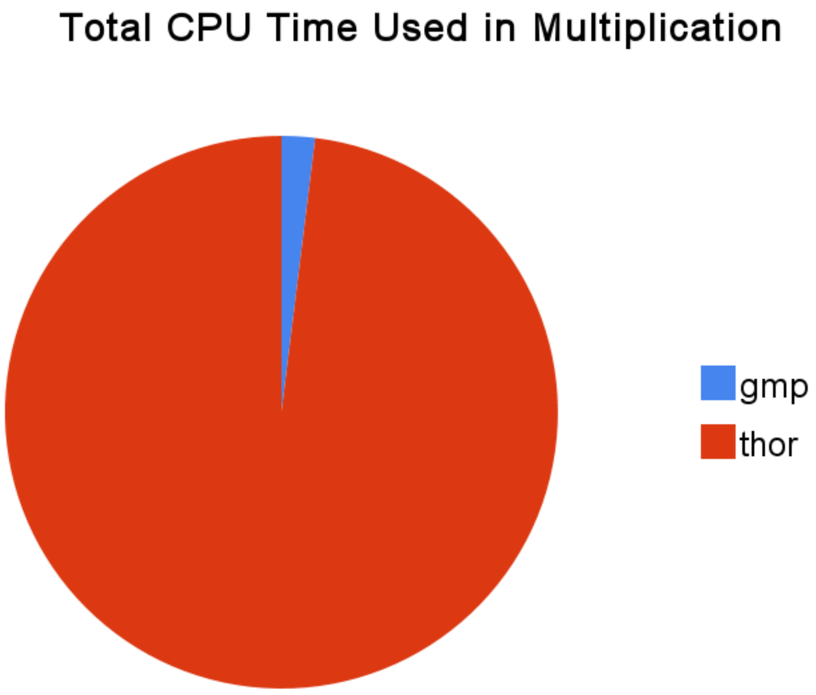
```
int vecthor_virtual_size(vecthor a){
    return spu_extract(*a,1);
}


int vecthor_actual_size(vecthor a){
    return spu_extract(*a,0);
}


int vecthor_sign(vecthor a){
    return spu_extract(*a,2);
}


void print_vector(vector signed int a){
    printf("%d ",spu_extract(a,0));
    printf("%d ",spu_extract(a,1));
    printf("%d ",spu_extract(a,2));
    printf("%d ",spu_extract(a,3));
    printf("\n");
}


void print_vecthor(vecthor a){
    int i,j,lim,hit=0;
    if(vecthor_actual_size(a)==0){printf("INVALID SIZE: reported size
is %d\n", vecthor_actual_size(a)); return;}
    if(vecthor_sign(a)<0){printf("-");}
    lim=abs(vecthor_actual_size(a));
    for(i=vecthor_actual_size(a)-vecthor_virtual_size(a)+1; i<=lim;
i++){
        for(j=0; j<4; j++){
            if(hit==1){printf("%08d",spu_extract(*(a+i),j));}
            if(hit==0 && spu_extract(*(a+i),j)!=0){hit=1;
printf("%00d",spu_extract(*(a+i),j));}
        }
    }
    if(hit==0){printf("0");}
    printf("\n");
}


void print_vecthor_debug(vecthor a){
    int i,j,lim;
    if(vecthor_actual_size(a)==0){printf("INVALID SIZE: reported size
is %d\n", vecthor_actual_size(a)); return;}
    if(vecthor_actual_size(a)<0){printf("-");}
    lim=abs(vecthor_actual_size(a));
    for(i=0; i<=lim; i++){
        print_vector(*(a+i));
    }
}


void vecthor_lightning_add(vecthor c, vecthor a, vecthor b){
    int i;
    for(i=VECS_IN_VECTHOR; i>=1; i--){
        *(c+i)=spu_add(*(a+i),*(b+i));
    }
}


void vecthor_add(vecthor c, vecthor a, vecthor b){
    if(c==a){c=aux[ADD_AUX];}
    vecthor_lightning_add(c,a,b);
    vecthor_carry(c);
    if(c==aux[ADD_AUX]){vecthor_copy(c,a);}
    return;
}


void vecthor_carry(vecthor c){
    vecthor_carry_ext(c,BIGGEST_INT);
}


void vecthor_carry_ext(vecthor c, int biggest){
    vecthor_write_virtual_size(c);
    int i,j,temp,lim=vecthor_actual_size(c)-vecthor_virtual_size(c);

    for(i=vecthor_actual_size(c); i>=lim; i--){
        for (j=3; j>=1; j--){
            temp=spu_extract(*(c+i),j);
            *(c+i)=spu_insert(spu_extract(*(c+i),j-
1)+temp/(biggest+1),*(c+i),j-1);
            *(c+i)=spu_insert(temp%(biggest+1),*(c+i),j);
        }
        temp=spu_extract(*(c+i),j);
        *((c+i)-1)=spu_insert(spu_extract(*((c+i)-
1),3)+temp/(biggest+1),*((c+i)-1),3);
        *(c+i)=spu_insert(temp%(biggest+1),*(c+i),0);
    }
    vecthor_write_virtual_size(c);
}


void vecthor_sub(vecthor c, vecthor a, vecthor b){
    int i,j,mask,flip,
    negate=0,
    a_size,b_size;
    vector signed int d;

    if(c==a){c=aux[SUB_AUX];}

    vecthor_clear(c);

    //special cases:
    if(vecthor_comp(a,b)==-1){vecthor e; e=a; a=b; b=e; negate=1;}

    a_size=vecthor_actual_size(a);
    b_size=vecthor_actual_size(b);
    for(i=a_size-vecthor_virtual_size(a)+1; i<=a_size; i++){
        for(j=3; j>=0; j--){
            //Perform normal subtraction (don't mind the
+spu_extract(*(c))...)
            *(c+i)=spu_insert(spu_extract(*(a+i),j)-
spu_extract(*(b+i),j)+spu_extract(*(c+i),j),*(c+i),j);

            //our mask variable is one that is used for absolute
value...it basically gives us the value of the sign bit.
            mask=spu_extract(*(c+i),j)>>31;

            //Splat the sign of the int in the vector to the auxilliary
vecthor.  (1=negative 0=positive)
            d=spu_insert(mask*-1,d,j);

            //Make all values in the res vecthor positive.
ABS(x)=(x+mask)^mask; (^ means XOR which means Exclusive OR).

 *(c+i)=spu_insert((spu_extract(*(c+i),j)+mask)^mask,*(c+i),j);

            /*
             *    Essentially if a value WAS negative (which we can
compute from the aux vecthor),
             *    then we 'flip' it (subtract it from the biggest
possible number).  I hold this in flip for ease of use.
             */
            flip=((BIGGEST_INT+1)*spu_extract(d,j))-
spu_extract(*(c+i),j);

            //redo the mask because we want to find abs() of something
else now.
            mask=flip>>31;

            //Replace all value in res vecthor with the flip (if
applicable).
            *(c+i)=spu_insert((flip+mask)^mask,*(c+i),j);

            /*    Finally, if a value WAS negative (which we can
compute from the aux vecthor [d]),
             *    Then we subtract ONE from the entry to the left of it
(which is why you see that weird
             *    +spu_extract(*(c+i),j) in the first line of the
loop.  We insert this into the res vec and
             *    then we're done!  Do it for all the others....
             */
            *(c+i)=spu_insert(spu_extract(*(c+i),j-1)-
spu_extract(d,j),*(c+i),j-1);
        }
    }

    if(negate){
        *c=spu_insert(-1,*c,2);
        vecthor e; e=a; a=b; b=e;
    }

    if(c==aux[SUB_AUX]){vecthor_copy(c,a);}
    vecthor_write_virtual_size(c);
}


extern vecthor aux[];
void vecthor_mul(vecthor c, vecthor a, vecthor b){
    int i,j;
    unsigned long long big_result;

    if(c==a){c=aux[MUL_AUX];}
    vecthor_clear(c);

    for(i=0; i<vecthor_find_max_pos(b); i++){
        for(j=0; j<vecthor_find_max_pos(a); j++){
            big_result=(unsigned long long) vecthor_pos_extract(b,i) *
vecthor_pos_extract(a,j);
            vecthor_pos_mul_add(c,i+j, big_result);
        }
        if(i%4==3){vecthor_carry(c);}
    }
    if(c==aux[MUL_AUX]){vecthor_copy(c,a);}
}


void vecthor_pow(vecthor c, vecthor a, vecthor bb){
    int b=spu_extract(*(bb+vecthor_actual_size(bb)),3);
    vecthor_clear(c);
    str_to_vecthor(c,"1");
    vecthor_copy(a,aux[POW_AUX]);

    while(b){
        if(b&1){
            vecthor_mul(c,c,aux[POW_AUX]);
        }
        vecthor_mul(aux[POW_AUX],aux[POW_AUX],aux[POW_AUX]);
        b>>=1;
    }
    vecthor_write_virtual_size(c);
}


void vecthor_mod(vecthor c, vecthor a, vecthor b){
    vecthor_copy(a,c);

    while(vecthor_comp(c,aux[MOD_AUX])==1){
        vecthor_lshift(aux[MOD_AUX],b,vecthor_bin_diff(c,b));
        vecthor_sub(c,c,aux[MOD_AUX]);
    }
    vecthor_sub(aux[MOD_AUX],aux[MOD_AUX],c);
    vecthor_copy(aux[MOD_AUX],c);
}


void vecthor_powm(vecthor c, vecthor a, vecthor bb, vecthor m){
    int b=spu_extract(*(bb+vecthor_actual_size(bb)),3);
    str_to_vecthor(c,"1");
    while(b){
        if(b&1){
            vecthor_mul(a,a,c);
            vecthor_mod(a,a,m);
        }
        vecthor_mul(a,a,a);
        vecthor_mod(a,a,m);
        b>>=1;
    }
    vecthor_write_virtual_size(c);
}


void vecthor_is_prime(vecthor c, vecthor a){
    int i;
    vecthor primelist[3],
    one,
    dec,
    d;
    d=init_vecthor();
    str_to_vecthor(primelist[0],"2");
    str_to_vecthor(primelist[1],"3");
    str_to_vecthor(primelist[2],"5");
    str_to_vecthor(one,"1");
    vecthor_sub(dec,a,one);

    i=0;
    //for(i=0; i<3; i++){
        vecthor_powm(d,primelist[i],dec,a);
        *(c+vecthor_actual_size(c))=spu_splats(!vecthor_comp(d,one));
    //}
}


int vecthor_comp(vecthor a, vecthor b){
    int i,size=vecthor_actual_size(a);
    //if(a_size!=b_size){return a_size>b_size ? 1:-1;}
    for(i=size-vecthor_virtual_size(a)+1; i<=size; i++){
        if(spu_extract(spu_gather(spu_cmpgt(*(a+i),*(b+i))),0)){return
1;}
        if(spu_extract(spu_gather(spu_cmpgt(*(b+i),*(a+i))),0)){return
-1;}
    }
    return 0;
}


int vecthor_cnt_lz(vecthor a){
    int a_z=0,lim,j;
    vector unsigned int a_zeros;
    lim=vecthor_actual_size(a)-vecthor_virtual_size(a);
    a_z=128*(lim);
    a_zeros=spu_cntlz(*(a+lim+1));
    for(j=0; j<4; j++){
        if(spu_extract(a_zeros,j)==32){a_z+=32;}
        else{a_z+=spu_extract(a_zeros,j); return a_z;}
    }
```

```
}


void vecthor_lshift(vecthor c, vecthor a, int times){
    int i,lim=vecthor_actual_size(a)-
vecthor_virtual_size(a)+1,size=vecthor_actual_size(a);

    while(times > 4){
        for(i=lim; i<=size; i++) {
            *(c+i)=spu_slqw(*(a+i),4);
        }
        times-=4;
        vecthor_carry(c);
    }

    for(i=lim; i<=size; i++){
        *(c+i)=spu_slqw(*(a+i),times);
    }

    vecthor_carry(c);
}


int vecthor_bin_diff(vecthor a, vecthor b){
    //a whole vecthor is shifting 107/108 times
    return vecthor_cnt_lz(b)-
vecthor_cnt_lz(a)+((vecthor_actual_size(a)-
vecthor_actual_size(b))*107);
}


int vecthor_pos_extract(vecthor a, int pos){
    int pos_in_vector=(3-pos%4);
    int vec_number=(32-pos/4);

    return spu_extract(*(a+vec_number),pos_in_vector);
}


void vecthor_pos_mul_add(vecthor a, int pos, unsigned long long x){
    int pos_in_vector=(3-pos%4);
    int vec_number=(32-pos/4);

    int pos_in_vector_big=(3-(pos+1)%4);
    int vec_number_big=(32-(pos+1)/4);

    int small_part=x%(BIGGEST_INT + 1);
    int big_part=x/(BIGGEST_INT + 1);


 *(a+vec_number)=spu_insert(vecthor_pos_extract(a,pos)+small_part,*(a+v
ec_number),pos_in_vector);

 *(a+vec_number_big)=spu_insert(vecthor_pos_extract(a,pos+1)+big_part,*
(a+vec_number_big),pos_in_vector_big);
}
```

```
int vecthor_find_max_pos(vecthor a){
    return 128-(vecthor_actual_size(a) - vecthor_virtual_size(a))*4;
}


void start_timer(void){
    spu_write_decrementer(0);
}


int read_timer(void){
    return -spu_read_decrementer();
}


double ticks_to_ms(int x){
    return ((double)x)/80000;
}


extern vecthor aux[];
void init_aux(void){
    int i;
    for(i=0; i<NUM_AUX; i++){
        aux[i]=init_vecthor();
    }
}


int add_elements(vector signed int a){
    return
spu_extract(a,0)+spu_extract(a,1)+spu_extract(a,2)+spu_extract(a,3);
}


int intpow(int x, int pow){
    int res=1;
    while(pow>0){
        res*=x;
        pow--;
    }
    return res;
}


int abs(int x){
    int mask=x>>31;
    return (x+mask)^mask;
}
```

Appendix Z - EPIC FAIL

The chart below uses actual data from our timing calculations. Apparently Thor was unsuccessful.



Clearly.