Reconfigurable Computing for High Performance Technical Computing

Troy Benjegerdes <troy@scl.ameslab.gov> and

Sean Stanek <vulture@scl.ameslab.gov>

Scalable Computing Lab

Ames Laboratory,

Ames, IA 50010

Introduction/Problem Background

Reconfigurable computing has proven to be quite successful at delivering improved performance in the form of speed-ups and power consumption for embedded and signal processing applications. Current trends in process technology and FPGA die size indicate that it will soon be possible to deliver similar kinds of performance improvements to high performance technical computing (HPTC) applications.

Many existing HPTC applications are characterized by requirements for 64 bit floating point, large, legacy code bases, and small number of experts in the field. Computational chemistry, fluid dynamics, and weather model codes are all examples of this type. Most, if not all of the major code bases have been parallelized over the last 10 years to run on tightly-couple clusters of commodity CPU's. Taking advantage of reconfigurable computing will either require parallelization at lower level, or tools to extract parallelism from the existing code. Requirements for 64 bit IEEE floating point arithmetic dramatically increase die size area requirements, making it harder to utilize techniques that have been successfully used for signal processing or embedded applications.

Goals/Objectives

Our goals for this project are to examine the GAMESS computational chemistry code as an example code. If a methodology can be developed to analyze and accelerate a 'hard' HPC problem like computational chemistry with reconfigurable computing, it may be applicable to other HPC code types.

First and foremost, any methodology for accelerating HPC codes like this is going to require collaboration with experts in the field. Fortunately for us, GAMESS is maintained by a group in the Iowa State Chemistry department. We also have been working with Brett Bode (who works on components of GAMESS), and Ricky Kendall, the author of large parts of the NWChem computational chemistry package from Pacific Northwest National Labs.

The next step is to profile the target application, and in collaboration with Dr. Kendall and Dr. Bode, develop a projection for possible speedups. Then we can determine candidate subroutines for execution on an FPGA. Once we have chosen a candidate function, we can determine function latency, pipeline depth, and FPGA area requirements.

Experimental Setup

From conversations with Dr. Kendall and Dr. Bode, it was determined that the most likely part of chemistry codes that can be accelerated is what is termed the 'integral generation'. Generating individual integrals is easily parallelized since each integral is independent of the others. However, current algorithms introduce a significant amount of control flow and optimizations to reduce the number of calculations required to generate a single integral.

We have estimated that a compact integral code will consist of around 5000 lines of Fortran code. Actual production chemistry codes such as GAMESS and NWChem have 10-20 thousand lines of integral code. Our estimates are that something like 10-20 lines of Fortran code could produce 30-40 multiplies, which is about the limit of what can be pipelined onto 1 large Virtex-II pro FPGA. Rewriting integral code appears to be the best option for accelerating chemistry apps for reconfigurable computing, but it will require several years of work in collaboration with experts in theoretical chemistry.

In light of this, we looked for another approach. The integral generation code uses a significant amount of transcendental functions, such as exp(), pow(), and sqrt(). We estimated that we could reasonably offload at least one transcendental function into a single FPGA. As FPGA area continues to increase in the future, this advantage will continue to grow.

The first task was then to get a profile of transcendental functions vs. calls to floating point multiply/add. This would then be used to choose a function to implement on an FPGA. There are several difficulties in doing this. First, the transcendental functions are in the C library's libm, which is normally not built with profiling support. Second, there would be no good way to determine whether a floating point multiply or add was part of computing a transcendental function or not.

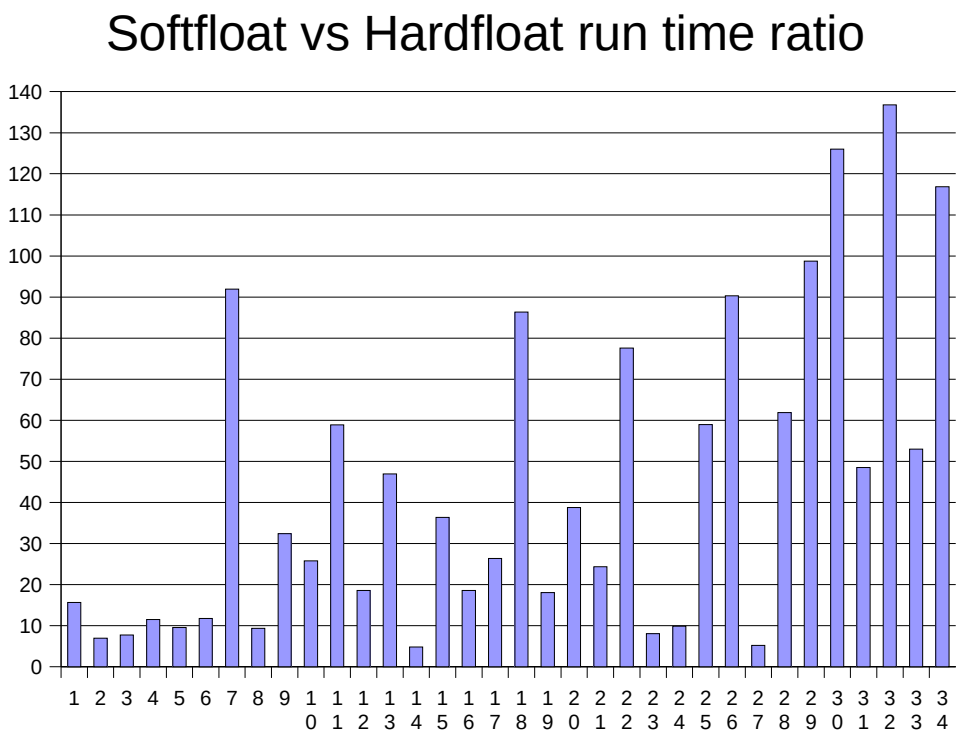
Our solution was to the software floating point features of gcc for PowerPC. This provides a secondary advantage of building a binary that could theoretically run on the PPC405 embedded in the VirtexII-Pro. This required rebuilding GCC, a C library (in this case uCLibc, http://www.uclibc.org), as well as GAMESS.

Results

The system used for profiling GAMESS was a Titanium Powerbook, with a 667 mhz PPC7455 (g4) running Debian Linux. GAMESS was first built with software floating point and profiling (gcc -msoft-float -pg). Building GAMESS itself took around an hour.

Next, a C library with software floating point and profiling support was needed. However, once uclibc was built (which took another hour), it was discovered that the g77 Fortran compiler has dependencies on the regular C library, which uses hardware floating point. On PowerPC, this causes invalid data to be used when a software floating point function calls a hardware float function. To resolve this required rebuilding gcc to get the 'libg2c' g77 Fortran library to work correctly with software floating point. This did, however provide the advantage of adding profiling support to all the software floating point code provided by GCC, so that every call to floating point multiplies and adds was profiled. Adding all the profiling and software floating point adds a significant overhead to running. The overhead resulted in a slowdown from 10 to 100 times compared to a GAMESS build with profiling using hardware floating point and an unprofiled C library, as shown in Figure 1.

Figure 1. Software floating point vs. Hardware floating-point ‘slowdown’



To gather profiling data, there were 34 example input declarations provided with the GAMESS source code. Each of these were run with both the hardware and software floating point to determine run time of transcendental functions vs. multiplies/adds, as well as determine which functions in GAMESS are commonly used.

After looking at the data, nothing really stuck out as taking a lot of time except the MAIN function and software floating point subroutines. GAMESS is a very large code base, and the code paths taken are very depended on the type of simulation being run. The ratio of software to hardware floating point in figure [1] gives some indication of how much the code is control flow vs. computation, with smaller ratios indicating more time spent outside of floating point computations, and this is very dependent on run type. There are also significant optimizations for avoiding computations on data that will not affect the outcome, which make different input data for the same type of run variable.

Figure 2: GAMESS % of time in various software floating point functions

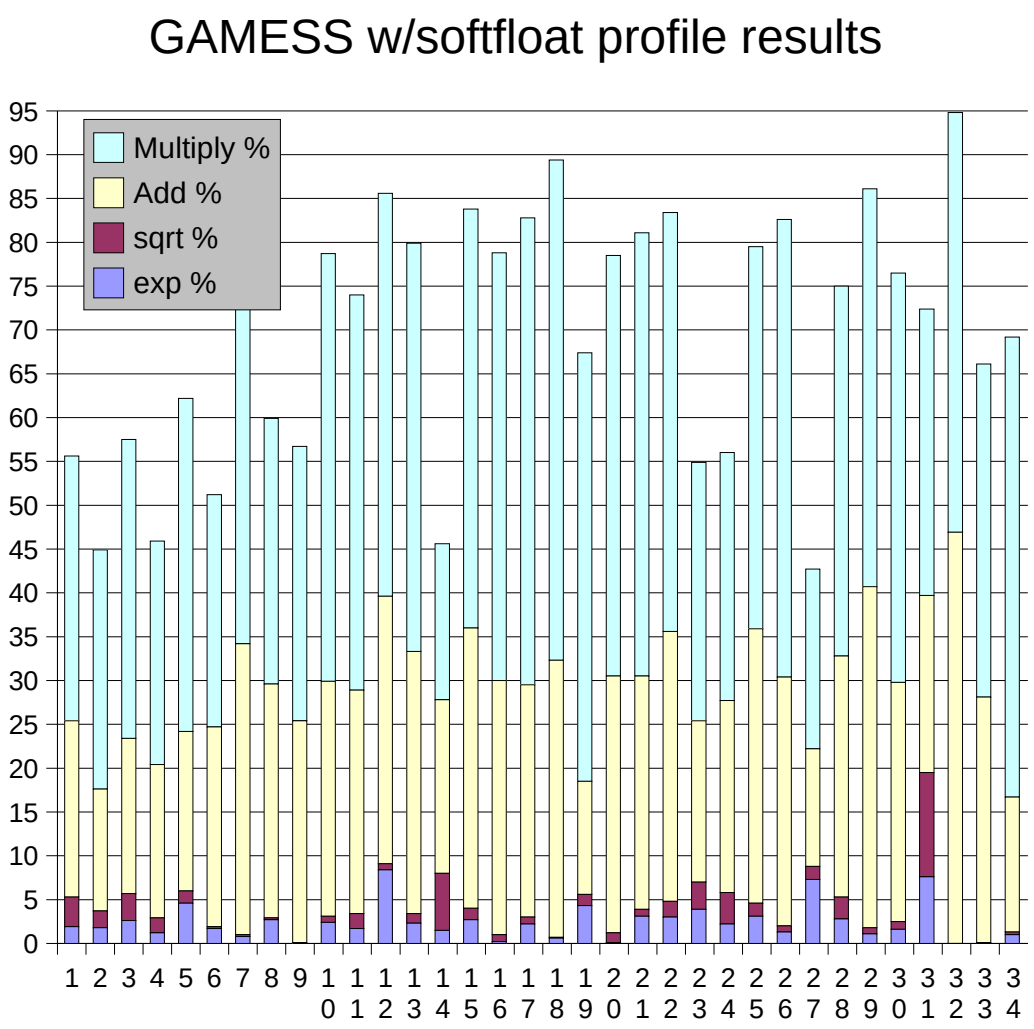


Figure 2 above shows the percent of time spent in each floating point function. The total graph height for each example run type (X-axis) is the sum of the 4 different floating point functions. This includes overhead of software floating point and profiling, so from a total application run time perspective, it may not be representative. However, the ratio of multiplies to other functions should be valid. The only transcendentals with significant (greater than 1%) time were sqrt() and exp(). Since some CPUs (ppc970) already implement sqrt() in hardware, we chose to implement exp() on an FPGA.

Based on power and total GFLOPS numbers for a matrix multiply in http://lotus1000.usc.edu/prasanna/papers/govinduHPEC03.pdf, we estimate that a pipelined exp () implemented on a VirtexII-Pro will show advantages over Opteron or PowerPC CPUS.

Implementation of Exp()

The basic implementation of the exp() function comes from the generic equation

$\exp(x) = e^x = 2^{x*\log_2 e}$.

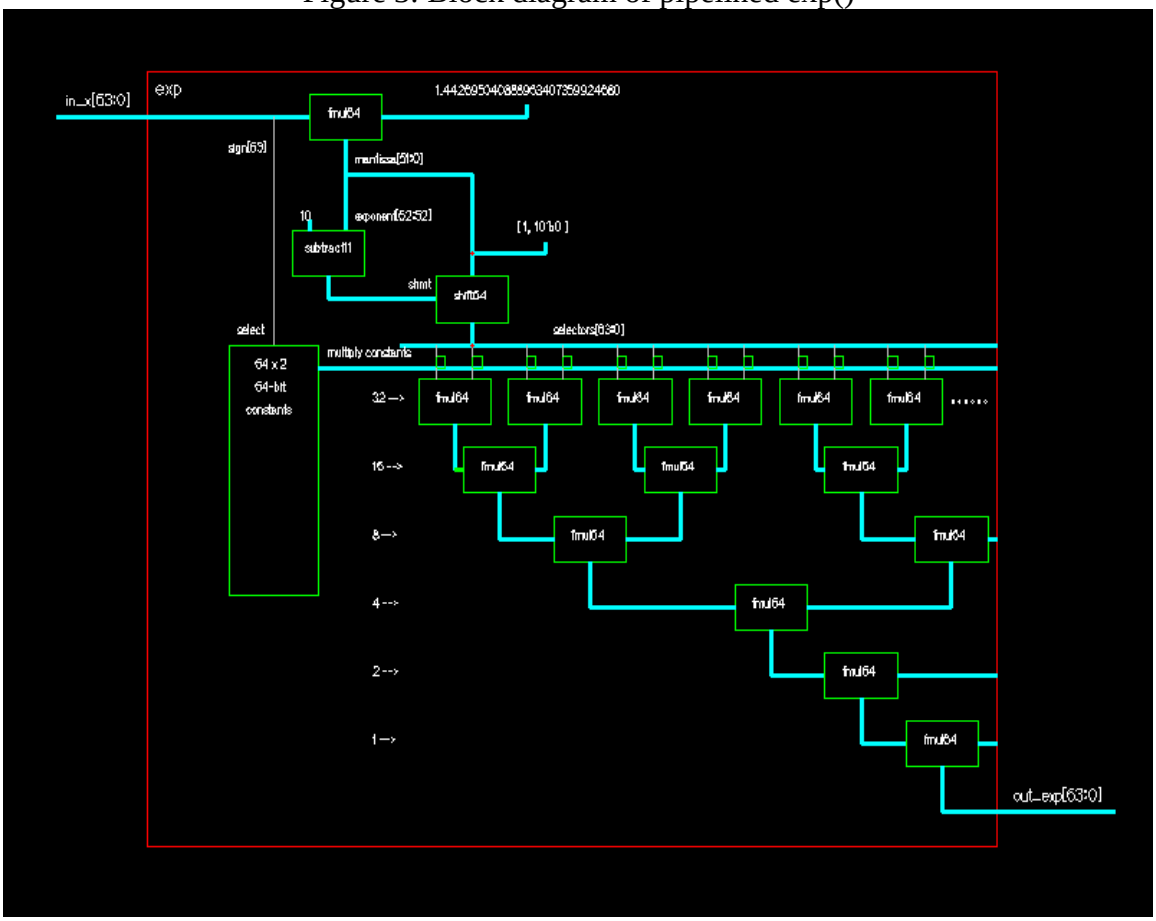
The x86 FPU itself has a hardware instruction to compute 2^x, so it is relatively fast in its computation of exp(). To make this type of hardware implementation feasible, an FPGA implementation should be at least as fast as a commodity microprocessor. It is fairly trivial to multiply x by a constant before using it as an exponent. Thus, the main implementation problem is quickly calculating 2^x where x is a floating-point value.

The implementation tested and chosen for this project is essentially a fully parallelized and pipelined method to calculate 2^x. We can split every bit in x up and multiply together individual bits of 2^x to get a final result. This is the simple multiplicative rule of powers, whereby one must multiply together two results to add two exponents under the same base. In other words, $2^x = 2^a + 2^b$. If x is split up into a separate bit for each bit in the mantissa for a floating point number, then 2^x can be calculated by a convolution of various constants 2^{-10}, 2^{-9}, 2^{-8}, etc. with each element selected depending on whether or not its mantissa bit was on or off. If the mantissa bit was off, then it is equivalent to instead multiply by 1.0 to perform a no-op. It is preferable to waste a multiplication in this implementation in order to avoid any possibility of complex control and routing logic. All the convolution constants are precalculated and stored in a table, which is constantly feeding data to the multipliers. Raising e to a negative power can be calculated by finding the multiplicative inverse of e to the same but positive power. This can be done by storing two versions of the precalculated data - the normal and the inverse of each power by 2.

The problem is now reduced to a convolution of up to 64 floating-point numbers. This is done in a tree fashion to reduce computation latency to $\log_2 n$ time. To include all of the mantissa would take a convolution of 52 numbers, however, because of the tree style convolution, it would take just as much time to convolute 64 numbers. Also, because of the way this is implemented, a special trick is used on the exponent field to shift the mantissa into place so that only a single variable shifter needs to be used on the floating-point number. The extra 12 multipliers are needed so that this is a very simple operation. The mantissa could have been shifted, and each position in the convolution receives a hardwired value for its 2^x, so control logic can be reduced by including the extra 12 multipliers. This reduces latency a little bit at the expense of about 23% increase in FPGA usage. This may be undesirable, but could perhaps be fixed by implementing that extra control logic.

The run-time of this implementation will take one initial multiply stage to calculate $x*\log_2 e$, then an add/shift stage (multiplies take a little longer than add/shift, so both of these can be done in the same stage), and finished by the 6-stage tree-style convolution, for a total of 8 stages in the pipeline. The multipliers could be implemented in different ways, in which case it could be possible that the actual pipeline is a hundred or so stages long. Depending on the way the multipliers are implemented, this algorithm will see a theoretical latency of 57-162 clocks, and a theoretical throughput of 130 to 200 million results per second. The amount of FPGA resources needed to implement this algorithm would require almost all of the CLBs on the largest currently available Virtex-II Pro part (55,616 slices). Figure 3 shows a block diagram of implementation.

Figure 3: Block diagram of pipelined exp()



Conclusion/Future Work

We have presented profile data and a proposed pipelined exp() implementation. Based on this information, it is unlikely we can achieve a significant speedup without either rewriting core loops to be pipelined, or using some tools to pipeline multiplies, adds, and exp.

Reconfigurable computing does appear to be a very promising technology. Future work on this subject would include the following:

- verifying correct operation of a pipelined-C code simulation of exp() on an FPGA in a computational chemistry app
- implementation of an exp() unit in verilog/vhdl and running on a VirtexII Pro part
- Eventually running a ppc-softloat build of GAMESS on the ppc405 and using the FPGA as a floating point unit
- re-implementation of chemistry integral code kernel for reconfigurable logic

References to come