263-2300-00: How To Write Fast Numerical Code

Assignment 2: 80 points

Submitted by Jinank Jain

Solution 1

Fractal Compression - Submitted on the Portal.

Solution 2

Some technical details about test setup:

- Compiler: clang-800.0.42.1
- Machine: MacOSX Intel(R) Core(TM) i7-3520M CPU @ 2.90GHz
- Processor Generation: Ivy Bridge

Part b:
In this part I tried four different variations which are discussed as below:

- In the first approach I just unroll the inner loop and does not do any modifications with that unrolled loop. The function is called `loop_unroll` in C file.
- In the second approach, I did some optimization with unrolled loop that I did in the previous section, I replaced $x[i]$ with some some scalar a which stores the value of $x[i]$ in some register. Another optimization was to much multiple accumulator instead single sum. I used for different sum variable to store the intermediate result. The function is called `scalar_replacement` in C file.
- In the third approach I unrolled the first loop too. Since in the array y there are only 5 values and I want them to be stored in registers so I unrolled first loop in multiple of 5 and storing all the values in the register. And creating 20 different sum accumulator and summing up the final answer. The function is called `super_scalar_replacement` in C file.
- In the fourth approach I remove the assumption which I took in third approach that n was divisible by 5 so for that I need introduce some branching instructions. The function is called `super_scalar_replacement_generalize` in C file.

Part d:

Discussions:
If we look at the Table 1 when we unroll the loop to the maximum extend when we put all the values of array y in the register we hit the maximum performance which should be the case as we would not any latency penalty by hitting cache or memory all the time. If we remove the assumption that n is

Optimization Flags	**slow performance1**	**loop_unroll**	**scalar _replacement**	**super _scalar _replacement**	**super_scalar _replacement _generalize**
-O0	0.164 FLOPs/c	0.238 FLOPs/c	0.233 FLOPs/c	0.817 FLOPs/c	0.738 FLOPs/c
-O3	0.512 FLOPs/c	0.506 FLOPs/c	0.511 FLOPs/c	1.338 FLOPs/c	1.112 FLOPs/c

Table 1: Performance comparison between different optimization function

a multiple of 5 we need to introduce some if conditions due to which there is going to be some extra comparison instruction which is going to affect the pipeline depending upon how good compiler do branch prediction. If we look at the first two case where we just do some loop unrolling and some scalar replacement we do not achieve much speed up from the standard implementation as we have discussed in lecture that doing just loop unrolling does not increase performance but it introduces a space where we could perform some optimization which I did in the super_scalar_replacement and super_scalar_replacement_generalize.

Speed Up:

- `loop_unroll`: 1.45, 0.98
- `scalar_replacement`: 1.42, 0.99
- `super_scalar_replacement`: 4.98, 2.61
- `super_scalar_replacement_generalize`: 4.5, 1.75

Differences in Speed Up with -O0 and -O3
Since we know that O3 has lot of advance optimization features as it is able to do optimizations like loop unrolling and all that stuff by itself that's why speed up in first two case `scalar_replacement` and `super_scalar_replacement` is not much because compiler is also doing those optimizations. But that is not the case with -O0 flags and that's why we see speed up when we use O0 flag.

But in the last case when we kind of do super loop unrolling and bring all the contents of array y into register and that point we see a significant speed up as compilers could not figure out such optimization due to various reason like memory aliasing and all that stuff.

Solution 3

Some technical details about test setup:

- Compiler: clang-800.0.42.1
- Machine: MacOSX Intel(R) Core(TM) i7-3520M CPU @ 2.90GHz
- Processor Generation: Ivy Bridge

While running the experiments for benchmarking multiplication and division operations I ran each test 1000000 so that I can overcome all the noise which is there because of some variable initialization and some extra summation operation that I had to perform to fool the compiler. In order to determine the latency loop was designed in such a way that there was a dependency which would break the pipeling of multiple instructions and for getting throughput we need pipelling effect so in the loop I create a bunch(6) of instructions which would fill the pipeline always. Few more things which need to kept in mind while conducting experiments since I am repeating all the operations 1000000 times problems become numerically unstable so we should be wise enough to choose our operands. Operands selection could be found in code.

Part a:
Reference Values are taken from the manual provided before: Link
If we look at the Table 2, I am almost able to reach the theoretical limit (Reference Value) in the Regular Case where I just multiply some bunch of random doubles or divide some numbers making sure that they stay in range and does not get eliminated as dead code by compiler. Main thing to note is result is pretty consistent with the theoretical values and which should be the case until and unless we are doing something really stupid.

Flags: `-O3 -fno-tree-vectorize -mno-sse4 -mno-sse3`

Instruction	**Reference Values**			**Regular Case**			**Special Case: Mul: 0x0000000000000001**			**Special Case: Div: 2.0**		
	Latency	**TPS**	**Gap**	**Latency**	**TPS**	**Gap**	**Latency**	**TPS**	**Gap**	**Latency**	**TPS**	**Gap**
MULSD	5	1	1	5.06	0.91	1.09	165.37	0.008	124.92	NA	NA	NA
DIVSD	10-24	.05-.12	8-18	20.06	0.07	14.06	NA	NA	NA	10.42	0.117	8.50

Table 2: Microbenchmarking for multiplication and division instruction

Flags: `-O3 -fno-tree-vectorize -mno-sse4 -mno-sse3 -ffast-math -funsafe-math-optimizations`

Instruction	**Reference Values**			**Regular Case**			**Special Case: Mul: 0x0000000000000001**			**Special Case: Div: 2.0**		
	Latency	**TPS**	**Gap**	**Latency**	**TPS**	**Gap**	**Latency**	**TPS**	**Gap**	**Latency**	**TPS**	**Gap**
MULSD	5	1	1	5.13	1.31	0.76	135.15	0.008	116.87	NA	NA	NA
DIVSD	10-24	.05-.12	8-18	19.02	0.38	2.63	NA	NA	NA	10.27	0.54	1.86

Table 3: Microbenchmarking for multiplication and division instruction

Part b:
In this case we are working with denormal values like 0x0000000000000001 which are very small. Hence the production of a denormal number is sometimes called gradual underflow because it allows a calculation to lose precision slowly when the result is small. Handling denormal values in software always leads to a significant decrease in performance. In extreme cases, instructions involving denormal operands may run as much as 100 times slower and that is what we are experience in our case too. Gap and Latency becomes unpredicted and increase by almost a factor 30.

Part c:
When we use 2.0 as a special division operand we need to keep in mind that our numerator is big enough for repeating the experiment 1000000 so that it does not become 0 and we do an no-op. For that I choose my numerator as 1e200 which is big enough for such computations. But the point to note is we are hitting the lower bound of latency which shows that there is some sort of optimizations that can be performed for operands like 2.0.

Part d:
In this case when we turn on mathematical optimizations `-ffast-math -funsafe-math-optimizations` we see some interesting stuff. If we look at gcc manual. The gcc manual says that this option "allows optimizations for floating-point arithmetic that (a) assume that arguments and results are valid and (b) may violate IEEE or ANSI standards." Essentially, what this means is that the compiler will take certain shortcuts in calculating the results of floating-point operations, which may result in rounding errors or potentially the program?s malfunctioning (hence the unsafe part of the name).Therefore throughput and latency of double precision divisions are reduced to values similar to floating point multiplications. Reason is that compiler replaces all division instructions with appropriate multiplication values, and therefore reduces total throughput and latency with cost of reduced precision.