Programming for performance on multi-core and many-core (2013)

Programming Assignment 1 (PA 1) **Due Date: 30th Jan 2013** **(50 points)**

(Note: You can use the gcc or the icc compiler for the problems in this assignment, although it is encouraged to use icc. The results may be different depending on the compiler used, so please report the compiler and the compiler flags that were used). The report should contain the outputs of each code snippet along with clear analysis of the code optimizations attempted and their effects.
You should check the correctness of the code after optimizations are done by comparing the results with results of the original code. Enlist all the (meaningful!) optimizations you have attempted.

Submitting
Keep the same folder structure you were given. Archive in tar format the PA1 folder. It should contain the following in that folder: Makefile,source files and report.pdf (you can modify the provided report/report.tex and create it with make report). After running make in the PA1 folder it should produce the executable files in bin folder. When completed, submit your archive on the github site.

1. (Loop Unrolling & Register spills) **[10]**
Evaluate when register spill occurs and what is the degree of unroll that results in better performance. Take the following code and try different degrees of unroll. You need look at the assembly code to find out what is the first degree of unroll that results in register spill -- look for the stack pointer (sp/rsp in x-86 INTEL). You also need to measure and report how performance changes as the degree of unroll (and register spills) increase. Keep increasing the degree of unroll until you see a significant performance degradation. (Compile the code with no optimizations on)

```
a = 0;
for (x=0; x<10000; x++)
  a += buffer[x];
// Unrolled code by 4 (you will need to try with different degrees of unroll)
a = a1= a2= a3= 0;
for (x=0; x<10000; x+=4) {
  a += buffer[x];
  a1 += buffer[x+1];
  a2 += buffer[x+2];
  a3 += buffer[x+3];
}
a=a+a1+a2+a3;
```

(follow code template given in "unroll.c").

Need to observe how the unroll factor changes with the ***data type*** of the "buffer" used in the above code. Remember that the main advantage of Unrolling is to fill the stall cycles created by the dependent instruction executions, load latency cycles.

Instructions: Need to be careful to disable the compiler auto rolling. The default optimization for icc is "O2", which does auto unrolling. To analyze the effect of manual unrolling, use the option "-unroll=0" to disable compiler auto unrolling.
Open icc manual (man icc), and look for the optimizations performance each optimization level, O1, O2, O3.

module load intel (to be able to access the "icc" compiler).
module load pgi   (to be able to access the "pgcc" compiler).
The -opt-report 1, -opt-report 2, -opt-report 3 gives optimization report of icc.
icc options : -fcode-asm , -fno-inline.
The -Minfo option with pgcc optimization report, other useful options with pgcc are -Munroll, -Mnounroll.

"module display modulename" gives the details about the modulename.
"module list" gives the list of modules (softwares) included in your path.

2. (Loop Unrolling, Duff's device) **[5]**
What does this code snippet do?

```
// to and from point to arrays
double *to, *from;
register int count;
{
    register int n=(count+7)/8;
    switch(count%8){
    case 0:     do{     *to++ = *from++;
    case 7:             *to++ = *from++;
    case 6:             *to++ = *from++;
    case 5:             *to++ = *from++;
    case 4:             *to++ = *from++;
    case 3:             *to++ = *from++;
    case 2:             *to++ = *from++;
    case 1:             *to++ = *from++;
            }while(--n>0);
    }
}
```

The above code is a refined implementation of a technique called ***loop unrolling***. **What may be some advantages of this implementation? What are possible disadvantages of loop unrolling?**

3. Branch misprediction **[10]**
Use the code in the file "*mispred.c*" for this problem. Compile the program and generate the following binaries:
- a0.out: icc –O1 –no-vec
- a1.out: icc –O2 –no-vec
- a2.out: icc –O1 –DFAST –no-vec
- a3.out: icc –O2 –DFAST –no-vec

Use "perf" tool to analyze the branch miss rate here.
perf stat -e branches -e branch-misses -e cycles -e instructions ./a0.out
perf list (gives the list of options for perf stat).
icc -S mispred.c (generates the assembly code, this assembly is much informative that the one generated by gcc, it has information about branch prediction rate of each condition, C code line number for each assembly line etc.).

a) Profile a1.out and a3.out with perf or (Perfexpert on TACC Longhorn), and measure the branch misprediction rate. Report the events you used to measure the misprediction rate.
(update: Only TACC Stampede has "perf" installed.)

b) Disassemble a0.out and a2.out. You can see the assembly code by using the command objdump (use switch –S to intermix source code with disassembly: objdump –S a1.out). For the intermixing of source code, you need to compile the code with -g option. Observing the objdump of object code is easier than the executable code, i.e., use test.o, which is generated after compilation (-c option), instead of the final linked version, test (-o option).

Search for the code corresponding to the function check_3odd in both a0.out and a2.out. You need to explain what the compiler did to generate the assembly code. You should discuss the number of branches the compiler generated in each case.

c) Disassemble a1.out and a3.out. Explain what is the difference between a0.out and a1.out. Similarly, explain what is the difference between a2.out and 3.out.

d) Run a0.out, a1.out, a2.out, a3.out and measure the execution times. Explain the reason of the different execution times of these codes.

Run this experiment on at least two machines that have different architectures such as one on Intel and the other on AMD.

4. Optimize **[5]**
Look at the code "conditional.c" and optimize the code to reduce the execution time when the code is compiled with the –no-vec option. Run the original code and the modified one and report the execution times. You need to show your optimized code version and have a short explanation on how you optimized the code. Need to check for the correctness of the code.

5. Data Dependences and ILP: Write a program called reduction1.c containing a reduction, similar to the example code given below. Write another program reduction2.c that performs the same computation but has been modified to increase the Instruction Level Parallelism.
Generate three executable files:
-- a0.out: compile reduction1.c with default compiler options
-- a1.out: compile reduction1.c using –no---vec
-- a2.out: compile reduction2.c with default compiler options
-- a3.out: compiler reduction2.c using –no---vec Run the four generated files
Report the execution times of each of these programs (You may need to have a relatively
large array to obtain a meaningful execution times) and explain the reason of these different execution times. You may have to look at the assembly codes to understand the difference in performance of the four versions. Please show the code you wrote for reduction2.c

```
a=0;
for ( x=0; x < 10000; x++)
a+= buffer[x];
```

6. How fast can you make it go ? [20]

(USE THE CODE IN DIRECTORY MakeitFast)

Download, extract and inspect the code. Your task is to optimize the function called *superslow* (guess why it's called like this?) in the file comp.c. The function runs over an n x n matrix and performs some computation on each element. In its current implementation, *superslow* involves several optimization blockers. Your task is to optimize the code.

Edit the Makefile if needed (architecture flags specifying your processor). Running make and then the generated executable verifies the code and outputs the performance (for this the op count is underestimated by $2n^2$) of *superslow*. Proceed as follows

a) Identify optimization blockers discussed in the lecture and remove them
b) For every optimization you perform, create a new function in comp.c that has the same signature and register it to the timing framework through the "*register_function*" procedure in *comp.c*. Let it run and, if it verifies, determine the performance.
c) In the end, the innermost loop should be free of any procedure calls and operations other than adds and mults.
d) When done, rerun all code versions also with optimization flags turned off (-O0 in the Makefile).
e) Create a table with the performance numbers. Two rows (optimization flags, no optimization flags) and as many columns as versions of superslow. Briefly discuss the table.
f) Submit your comp.c to the github.

What speedup do you achieve?

Some Extra Problems [20]

E1. Measure the MFLOPS of several codes

In this problem you need to measure the MFLOPS of a few code versions when running different floating point operations (declare your variables as float).

Code 1: Measure the MFLOPS of a code that computes the sum of two matrices A and B and stores the result in C:
C[i][j] = A[i][j]+B[i][j];
Code 2: Measure the MFLOPS of a code that performs a matrix-matrix multiplication.

For each code version, you need to show the code you wrote, the measured MFLOPS, and explain how you computed them. For each code version you should try different matrix sizes. If your matrices are small, you may have to include an outer loop to increase the running time of your code. You should discuss if you observed any variation on the number of FLOPS as you changed the sizes of the matrices.
Note: you can use icc compiler and higher optimizations. Observe how the MFLOPs is varying for the two codes with respect to the size of the problem.

E2. Read as well as run the code to find out what the code is actually doing:

```
#include <stdio.h>

 int main( int argc, char **argv )
 {
   float machEps = 1.0f;

   printf( "current Epsilon, 1 + current Epsilon\n" );
   do {
     printf( "%G\t%.20f\n", machEps, (1.0f + machEps) );
     machEps /= 2.0f;
     // If next epsilon yields 1, then break, because current
     // epsilon is the machine epsilon.
   }
   while ((float)(1.0 + (machEps/2.0)) != 1.0);

   printf( "\nCalculated Machine epsilon: %G\n", machEps );
   return 0;
 }
```

E3:

You have N = 10^6 two-dimensional coordinates, and want their centroid. Which of these is faster and why?

1. Store an array of (xi ; yi) coordinates. Loop i and simultaneously sum the xi and the yi .
2. Store an array of (xi ; yi) coordinates. Loop i and sum the xi , then sum the yi in a separate loop.
3. Store the xi in one array, the yi in a second array. Sum the xi , then sum the yi .

Use the code ***mean.c***; If you use high optimization -O3, the compiler may optimize away your timing loops. This is a common hazard in timing. So, just compile with -O2.

E4: Why the code ***access.m*** performs the way it is ? How can you optimize the MATALB code to scale linearly.
Note: GNUPLOT command: gnuplot < nice_web_plot.gnu, generates a pdf file with the plot.