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

Programming Assignment 2 (PA 2)

Submission Guidelines:
Please maintain the same folder structure that was provided. Archive in tar format the PA2 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 PA2 folder it should produce the executable files in bin folder. When completed, submit your archive on the github site.

Instructions: *This is a group assignment and therefore there should be a single submission for the group, with the following naming convention, name1_name2_pa2.tar. All the groups should work on all the problems in Part A. The number of problems you can solve in Part B is optional; however, each group should solve at least 3 problems (4A to 4J).*

Part A:

1. SAXPY:

SAXPY is a BLAS 1 function that computes $y = \alpha x + y$ on single precision floats, where x, y are vectors and α a scalar. The file saxpy.c contains the scalar code for computing this operation in the function saxpy. Your task is to implement the a vectorized version of this procedure, called vec_saxpy in the same file, using SSE intrinsics. You can assume all arrays are 16-byte aligned. Compile using the provided makefile and run the code.
Compute the vectorization efficiency.

Vectorization efficiency (Veff) as the ratio is defined = Op count scalar code / Op count vectorized code;

Where the op count of the vectorized code includes all those instructions that perform operations and that are used to manage the positioning of the data in the registers (shuffle, unpack, etc…) with the exception of loads and stores. Note that certain intrinsics create a sequence of two or more instructions.

2. MVM Puzzle:

Your task is now to vectorize the code for a 5 x 5 MVM y = Ax: Implement the function vec_mvm5 in mvm5.c using SSE intrinsics; do not use any unaligned load or store instruction. You can assume all arrays are 16-byte aligned. Compile, run, and determine the vectorization efficiency.

Note: In all the above exercises you need to achieve a high enough vectorization efficiency to obtain full points. Always verify the code using the provided routines.

3. Measure the frequency of the processor:

Write a program to measure the frequency of the processor. Run your program and measure how close/far the frequency measured with your program is from the real frequency of the processor. You can find the frequency of the processor by running the command “cat /proc/cpuinfo”.
Hint: You can use the time stamp counter of the processor and gettimeofday(), but other mechanisms can also be used.
You need to report the code you wrote and the discrepancy between the measured frequency and the real one. You can run your code in different machines (that have different running frequencies).

Part B:

4. Compiler Vectorization:

In this section, you will be given a set of C codes and will ask you to compile them and go over the reports generated by the compiler. These codes contain simple loops that have been designed to challenge the vectorization capabilities of the compiler. As we will see, some of these codes are trivially vectorized by the compiler, while others are not. The goal of this section will be to teach why the compiler fails to vectorize these codes and give you some guidance about compiler transformations that can be used so that the compiler vectorizes them. We will also see cases that need to be manually vectorized, as the compiler does not vectorize them, no matter what transformations we apply.

For this section, we will use the Intel icc compiler. To compile these codes you need to run the command line

icpc –vec-report1 file.c

When using the option –vec-report1, the compiler generates a report that indicates the line number of the loops that were vectorized. When using the option –vec-report2 or -vec-report3, generates a more detailed report that includes the loops that were not vectorized and the reason for that.

Excercises

Topic 1: Undesirable transformations

4A. In the exercises below, we use an outer loop to increase the execution time of the loop whose vectorization we want to study. In all cases, we preceded this outer loop with a #pragma novector so that the compiler will not transform this artificial loop that we only inserted to improve the accuracy of our measurements. Your assignment is to explain what happens if the #pragma novector is removed. You can use t0.c to study this problem.

Topic 2: Non-unit strides

4B. **Effect of non-unit strides on compiler evaluation of profitability.**

Compile t1.c. Determine if the loop in line 12 is vectorized (notice that the loop induction variable of this loop is incremented by 2).

This loop has the form:

```
for (int i = 0; i < 1024; i+=2)
      A[i+1] = A[i] + B[i];
```

i. If not, transform the program so that the loop is vectorized.
ii. compute speedup.
iii. If you don't obtain speedup, Can you find a more complex right-hand side expression so that speedup is obtained when the loop is vectorized.

4C. Inlining and vectorization

i. Compile (t8.c). Notice that this function is called twice. In the first call to function t1, the last parameter is 1, while in the second one is 2.

```
#pragma auto_inline(off)
void t1 (float* __restrict__ A, float* __restrict__ B, int k)
{
      …
      for (int i = 0; i < 1024; i++) {
      A[i*k] += B[i];
      }
      …
}
int main() {
      …
      t1(A,B,1);
      t1(A,B,2);
      …
```

```
}
```

ii. Study what difference the use of #pragma auto_inline(off) when applied to function t1() makes.
iii. Explain

Topic 3: Program transformations
***4D. Effect of manual unrolling*.**
(a) Compile t2.c which contains the following loop:

```
for (int i = 0; i < 1280 - 5; i+=5){
 A[i] = B[i] * A[i];
 A[i + 1] = B[i] * A[i + 1];
 A[i + 2] = B[i] * A[i + 2];
 A[i + 3] = B[i] * A[i + 3];
 A[i + 4] = B[i] * A[i + 4];
 }
```

(b) Is this loop vectorized by the compiler? If not, rewrite the loop to enable vectorization.
(c) Compute speedup

4E. Induction variables.
a. Compile (t15.c) which contains the loop

```
k = 0;
for (int i = 0; i < 1024; i++){
A[k] = B[i] + 1;
k++;
}
```

b. Is the loop vectorized?
c. If so, what speedup do you obtain? If not, how do you need to transform the loop to enable vectorization?

4F. *Wrap-around variable*s.
a. Compile (t14.c) which contains the loop k = 0; j = 1;

```
for (int i = 0; i < 1024; i++){
A[k] = B[i] + 1;
k = j;
j++;
}
```

b. Is the loop vectorized?
c. If so, what speedup do you obtain? If not, how do you need to transform the loop to enable vectorization?

4G. Datadependence analysis:
a. Compile t3.c which contains the loop.

```
for (int j = 0; j < 512; j++) {
for (int i = 0; i < j; i++) {
AA[i][j] = AA[j][i] + BB[i][j];
}
}
```

b. Is the loop vectorized?
c. If not, try different ways to allocate the arrays, the restrict keyword or the #pragma ivdep to help the compiler vectorize this code. You may need to use the #pragma vector always to force the compiler to vectorize.
d. Explain the result you obtain.

4H. Datadependence analysis:
Repeat the previous exercise but now using program (t13.c) which contains the loop

```
for (int i = 1; i < 512; i++) {
for (int j = 1; j < 512; j++) {
A[i][j] = A[i-1][j] + A[i][j-1];
}
}
```

What happens in this case if you use the #pragma ivdep?

Topic 4: ***Backward traversals***

4I. ***Simple case of backward traversal***

a. Compile program (t9.c) which contains the loop:

```
for (int i = 1023; i >= 0; i--){
A[i] += B[i];
}
```

b. The compiler needs to reverse the loop to vectorize it.

c. Does the compiler vectorize the loop ? Explain what happens.

4J. ***Effect of reverse traversal on efficiency of vectorization*****.**

a. Compile program (t6.c) which contains the loop.

```
for (int i = 0; i < 1024; i++)
A[i] = (B[1023-i]*B[1023-i]-B[1023-i]);
```

b. Does the compiler vectorizes this loop ?

c. If not, transform the program to achieve vectorization.

d. Compute speedup.

Note: Problem 4 code snippets are taken from the Extended Test Suite for Vectorizing Compilers: http://polaris.cs.uiuc.edu/~maleki1/TSVC.tar.gz

This test suite contains a comprehensive list of simple benchmarks for testing the efficiency of vectorization capabilities of the compiler.