CS/ECE 347, Spring 1998

Lab Assignment 2

Optimizing C Code

Assigned: Thursday, 26 February
Due: Thursday, 12 March

Introduction

In this lab, you will use the information presented in lecture 8, and your knowledge of the Alpha's execution pipeline to optimize a matrix multiply routine. You must use one of the class's Alphas to do this assignment.

Methodology

You will refine an implementation of matrix multiply increasing its speed with every revision. The file `main.c` contains a test bed for timing, and checking your multiply routines. The file `multiply.c` contains a skeleton of the hand in file. You will modify `multiply.c` to contain your answers.

Timing Using the Cycle Counter

The architecture of the Alpha requires that each implementation have a cycle counter register. The processor increments this register every clock cycle. The cycle counter register is comprised of two counters; the lower 32 bits are the number of cycles for the current process, and the upper 32 bits are used to calculate wall clock time. The cycle counter is very convenient for timing software, because it only takes one clock cycle to read the cycle count. We provide a function that reads the count register called `counter()` for timing routines in this lab. The `counter()` function has very low over head. For example, the value for `Delta` after executing the following code is 12 cycles.

```
Delta = counter();
Delta = counter() – Delta;
```

Cycle Counter Limits

Since the cycle counter is 32 bits wide, the counter rolls over about every 9 seconds. If the counter rolls over while you are timing a routine, the difference in times is negative. To work around this problem, the code throws out any negative elapsed times.

Matrix Constraints

The file `array.h` contains the interface for a variable size, two dimension array package, implemented by `array.c`. The files `matrixutil.{c,h}` contain a set of matrix utilities. The files `multiply.{c,h}` contain the interface specification for your answers, and the baseline implementation of the matrix multiply routine called `Multiply`. Although the routines in the array, and matrix utility modules can handle any size array, we are going to limit ourselves to matrices that are square, and whose height and width are always multiples of 16.

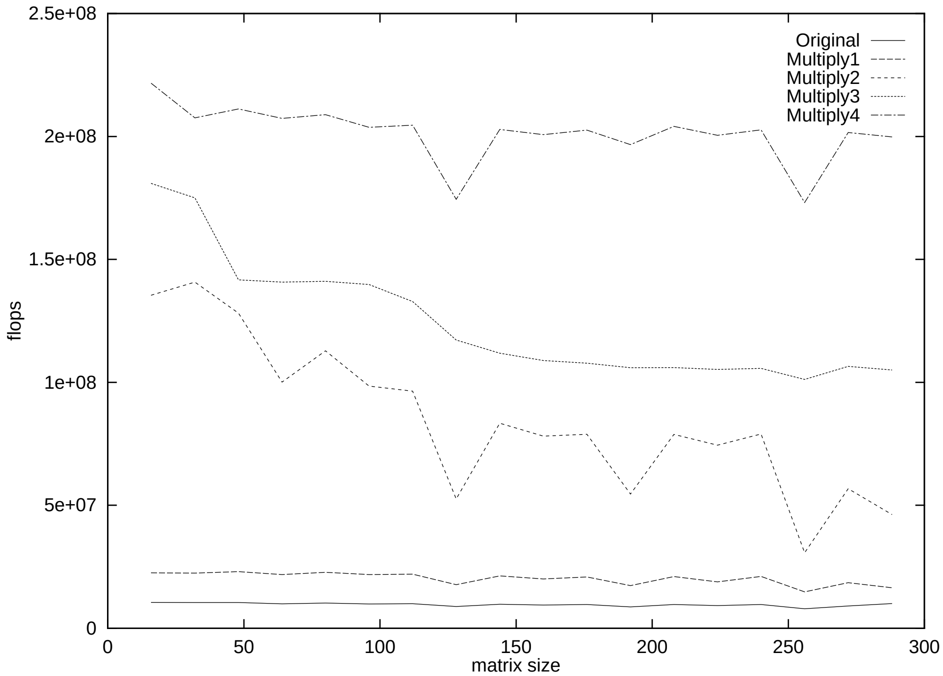
Acceptable Error

Every entry in the result of your matrix multiply routine must be within an absolute error of 0.000001 of the baseline implementation. The test bed checks that your multiply routine returns results with an acceptable error.

Visualization

The test bed program produces a file called `data`, which is a table of numbers suitable for importing into a spreadsheet or `gnuplot`. The program prints the appropriate commands for gnuplot to plot the data file at the end of each run. You can cut and paste these commands to graph your routine's performance. Use these plots to measure your progress.

To guide your optimization efforts, we have included a file baseline.dat that contains the performance numbers for the solution routines and has the same format as the data file from the test bed. To view the baseline file in gnuplot, issue the command load "baseline".



Baseline Performance 1

Logistics

For this lab, you may work in a group of two or three. The hand in for this lab is completely electronic. There is no hard copy to hand in. You should hand in one solution set for the whole group.

Directory Access

Be sure that the directory is not readable by the general public. You can list access ritghts to a directory by using the following command:

```
/usr/afs/bin/fs la
```

If you have an entry system:anyuser rl or system:authuser rl then anyone can read and list your directory. You can remove access writes for system:anyuser and system:authuser with the following commands:

```
/usr/afs/bin/fs sa -dir . -acl system:anyuser none
/usr/afs/bin/fs sa -dir . -acl system:authuser none
```

Unpacking

To get started, copy everything in

```
/afs/cs.cmu.edu/academic/class/15347-s98/public/labs/lab2
```

into your personal directory. After you copy the files into your directory, edit the file multiply.c, and add your personal information into the structure LabGroup. You will hand in the multiply.c , and the array LabGroup will be used to assign grades, so this array must be filled in accurately. You must use one of the class's Alphas to do this assignment!

Hand In

To hand in your work, rename the file `multiply.c` to `<andrewID>multiply.c` where <andrewID> is the andrew ID of one of the members in your group. Place this file in the directory

`/afs/cs.cmu.edu/academic/class/15347-s98/public/labs/lab2/handin`

Your group should do this before the date and time the assignment is due. You have only write and list permission for this directory. Be careful not to overwrite anyone else's solution.

Grading

Your `<andrewID>multiply.c` file is expected to implement the `multiply.h` interface so it can be compiled and linked with the testing program. If your group's file does not compile, 20 points will be deducted from your group's score. Each of the solutions to the problems will be timed, and checked for correctness. We expect you to be within 10% of the performance of the sample solution. The score of each problem is a combination of execution time, the number of correct multiplies, and the implementations of your routines. Its not enough that your code is fast and gives correct answers, your source code must follow good coding practices.

Remove Abstraction

The baseline code is implemented in a modular way. The module `array.{h,c}` implements a two-dimensional variable size array. The module `matrixutil.{h,c}` implements a bunch of matrix utilities based on the array module. The baseline `Multiply` routine in `multiply.c` strictly adheres to the `array.h` interface. Therefore, operations that would normally be a single instruction now require a function call.

Problem 1

Remove the overhead of abstraction by exposing array's implementation. Make a copy of the `Multiply` routine, and rename it `Multiply1`. Remove the abstraction overhead by inline expanding the routines from `array.c`.

Remove Optimization Blockers

Several C programming idioms force the compiler to be overly conservative generating instructions. Using a complex bound check in a `for(;;)` loop forces the compiler to produce inefficient code.

```
for( I = 0; I < a->height; I++)
```

In the above code, the expression `a->height` must be evaluated each time through the loop. This forces the compiler to deference the pointer a each time through the loop.

Another construct that blocks optimization opportunities is using an lvalue which references memory. Expressions like

```
C[q] += A[I] * B[I];
```

force the compiler to use `C[I]` as an accumulator instead a register. The compiler must assume that the three arrays A, B and C can overlap and affect each other, and therefore stores to `C[q]` could effect the values of `A[I]` and `B[I]`.

Problem 2

Remove the optimization blockers from the routine `Multiply1`. Make a copy of `Multiply1`, and rename it `Multiply2`. Add temporary variables to remove C constructions that block optimization.

You should disassemble your `Multiply2` routine to make sure that it is using a register as an accumulator in the inner loop. To view the instructions of your routine, execute the program dis using the following command line:

`dis –p Multiply2 multiply.o`.

Notice that the inner loop has been unrolled several times.

Instruction Latency and Change Indices

Use the information provided in lecture 8 to rearrange the index order to make the multiply use the cache more efficiently. Doing this will introduce a subtle performance problem, so this version of multiply maybe slower than Multiply2. To make this version of multiply faster than Multiply2, you need to remove the performance problem by rearranging expressions to reduce the effect of instruction latency.

An instruction's latency is the number of cycles for processor to produce a result. In general, all the floating point instructions have a latency of 4 cycles. An instruction A depends on an instruction B if A uses the result of B. Instructions often stall waiting for the results of instructions they depend on. Here is a short code sequence of code with bad dependency problems:

```
ldt       $f11, (t12)
ldt       $f12
mult      $f0, $f11, $f11
addt      $f12, $f11, $f11
stt       $f11, (t12)
```

In the above code, the `mult`, and `addt` instruction have a bad dependency. The `addt` instruction must wait 4 cycles for the result of the `mult` instruction. Adding instructions between the `mult` and the `addt`, would make the code more efficient. You can use `dis` to look for bad dependencies in the inner loop of your multiply routine.

Problem 3

Rearrange the indices of the multiply routine, and remove latency dependencies. Make a copy of `Multiply2`, and rename it `Multiply3`. Change the order of the indices for the matrix multiply to increase `Mutiply3`'s performance. This will cause a dependency chain problem. Use temporary variables and expression rearrangement to reduce the effect of instruction dependencies.

Blocking

The performance of `Multiply3` drops off as the matrices get bigger due to cache contention. Processing the multiply in blocks reduces the effects of cache contention. Use the information from lecture 8 to produce a blocking version of matrix multiply.

Problem 4

Write a matrix routine that uses cache blocking and is faster than `Multiply3`. Name this routine `Multiply4`. The code in lecture 8 is only a starting point. You need to apply the techniques from the previous three problems to make `Multiply4` faster than `Multiply3`.