"It's hardware that makes a machine fast. It's software that makes a fast machine slow." – Craig Bruce

Learning Objectives

Performance optimization and cache conscious programing, including

1. Analysis of cache access patterns
2. Loop tiling / strip mining
3. Single pass vs. multi-pass algorithms
4. Software prefetch insertion

Work that needs to be handed in (via SVN)

1. `transpose.cpp` – **this is due by the first deadline**
2. `filter.cpp` – **this is due by the second deadline**

Guidelines

- Your goal for this lab is to transform the provided code to reduce the execution time.
- All your code should be in the files indicated above. DO NOT change the function invocations that we provide, as those are the interfaces that our test code will use to verify the correctness of your code.
- Our test programs will try to break your code. Your code should produce the same output as the one generated by the code that we have provided. Thus, you are encouraged to create your own test scenarios to verify the correctness of your code.
- We will be testing your code using the `EWS Linux machines`, so we will consider what runs on those machines to be the final word on correctness and performance. Notice that computers differ in the architectural features they have. Thus, a program transformation that has a significant impact on one architecture may have little impact on another one, or vice versa. We have tested these code in a variety of machines and have observed a significant reduction in execution time after optimizing them with the transformations that we are asking you to implement. However, you SHOULD verify that your transformed code reduces the execution time in the `EWS Linux machines`.
- Since these are experimental results collected on a complex systems (potentially with other users simultaneously running jobs), you may need to run your code multiple times to get useful run times. In general, you should focus on the **shortest** execution times, since (unlike in other sciences) all of the variance is coming from interference which is slowing down your runs (i.e., the fastest times are the ones with the least interference, which is what we want.)

Matrix Transposition [20 points]

The function `transpose_none` (in file `main-transpose.cpp`) implements the transpose of a 2D-matrix.

```
for (int i = 0; i < SIZE; i ++) {
    for (int j = 0; j < SIZE; j ++) {
        dest[i][j] = src[j][i];
    }
}
```

In this code, accesses to `dest` have spatial locality as the matrix is accessed by rows; however, accesses to `src` do not have spatial locality, as it is accessed by columns. Thus, your goal for this lab is to transform the code above to increase locality in the accesses to `src`.

If you consider iterations `i=0` and `i=1` you can see that the rows of `src` that were accessed during iteration `i=0` will also be accessed during iteration `i=1`. Thus, all the memory accesses to `src` during iteration `i=0` should *miss*, while accesses during iteration `i=1` should *hit* (as cache lines contain several bytes of data) if the cache is large enough to hold the cache lines with the data from `src` that were brought to the cache during the execution of iteration `i=0`. However, if `src` is very large, the cache lines from `src` brought to the cache during the execution of iteration `i=0` will not fit in cache, and all the memory accesses to `src` during iteration `i=1` will result in a *miss*.

Your task is to apply loop tiling (see the wiki page below) to increase the likelihood that the cache lines in `src` remain in cache until they are reused (and result in a cache *hit*). This transformation is the same as the blocking transformation applied to optimize the performance of matrix-matrix multiplication during the class lectures.

`http://en.wikipedia.org/wiki/Loop_tiling`

Note: Explore the tile size that makes the code to run the fastest. Depending on what machine you run on, you should be able to cut the execution time significantly. (Notice that tile size is a parameter of the loop tiling transformation and is different from SIZE, which defines the SIZE of the images.)

IMPORTANT: This program uses a large amount of memory, so you'll need to run it in an environment with sufficient memory – a VM most likely won't cut it, but the EWS systems should work fine. Another consequence is that if your program segfaults and core dumps, the core dump files will be huge and can easily fill up your disk space, so you should disable core dumps:

```
ulimit -c 0
```

Filters [80 points]

Our second example program in function `filter_none` (file `main-filter.cpp`) is meant to represent a series of filters (`filter1`, `filter2`, `filter3`) applied to an image. Your task is to apply a series of transformations to reduce the execution time of this code. You should first run the program without optimizations (`./filter none`) to obtain the baseline performance of the the function `filter_none`. Then, you should modify the code according to the transformations described below, run the code and obtain the new execution time. You need to verify that the transformed code is correct (returns the same result obtained with `filter_none`) and the execution time is reduced. All the code modifications should be done in the `filter.cpp` file.

Note: You will see that `./filter none` prints "Image 2323". This is only the value of a *single random pixel*. This means that if you *don't* see "Image 2323" after making a transformation, you're *definitely* wrong, but if you *do* see "Image 2323", it doesn't mean that you're correct. You should modify `main-filter.cpp` to verify your final result in a fashion similar to what you did in the first part.

The transformations you need to apply are the following:

Prefetch

As discussed in lecture, hardware prefetchers are effective for streams and strided accesses, but typically can't effectively prefetch irregular access patterns. The provided code uses an array of pointers. Because we're walking down the array linearly, the hardware prefetcher should effectively prefetch the array itself, but it won't be able to prefetch the pixels pointed by the array. To prefetch the pixels, you'll need to use software prefetches.

In the last page of this handout, you'll find information about how to insert prefetches. Your code should include prefetches for each traversal of a pointer array. Make sure you are prefetching the pixels and not just the array! Explore the parameter space of this prefetching, including: (1) how far ahead should you prefetch? (e.g., should you prefetch 1 iteration ahead? 10? 50?), (2) does performance change if you prefetch for reading vs. writing? In which cases should you do each? and (3) how should you set the locality argument? You should be able to achieve non-trivial speedups, so don't quit until your code is noticeably faster.

Important remarks

- Your code with prefetch should be in the function `filter_prefetch`.
- You can execute the code with the command `./filter prefetch`.

Loop Fusion

One shortcoming of the supplied code is that, it traverses the images several times, once for each filter. As a result, we're bringing all the data through the cache several times, and each time only performing a relatively small amount of processing on it. There is an optimization called *loop fusion*, whereby we merge adjacent loops for efficiency sake, including cache efficiency. Refer to `http://en.wikipedia.org/wiki/Loop_fusion` for a little insight into loop fusion.

Transform the code `filter_none` to fuse the loops to decrease the number of walks through the arrays. Note that you will need to do a small amount of work outside of the fused loop to make this possible.

Important remarks

- Loop fusion is not always a legal transformation, meaning that sometimes it cannot be applied, as it will produce incorrect results. Whether a transformation is legal or not can be determined based on the memory accesses to the data. Thus, while applying this transformation to improve performance you need to verify that your code produces the same output as the original code that we have provided.

- Your code with loop fusion should be in the function `filter_fusion`.
- You can execute the code with the command `./filter fusion`.
- To get full credit, you will have to fuse all three loops. Fusing two loops will give you partial credit.

Loop Fusion and Prefetch

Generate a version of your code with loop fusion *and* software prefetching.

- Your code should be in the function `filter_all`.
- You can execute the code with the command `./filter all`.
- To get full credit, you will have to fuse all three loops. Fusing two loops will give you partial credit.

GCC support for prefetching

The GCC compiler provides the following builtin function (which gcc recognizes but isn't part of the C language) to provide prefetching. This interface closely matches x86 prefetch instructions. Below is excerpted from the GCC documentation.

Built-in Function:

```
void __builtin_prefetch(const void *addr, ...)
```

This function is used to minimize cache-miss latency by moving data into a cache before it is accessed. You can insert calls to `__builtin prefetch` into code for which you know addresses of data in memory that is likely to be accessed soon. If the target supports them, data prefetch instructions will be generated. If the prefetch is done early enough before the access then the data will be in the cache by the time it is accessed.

The value of `addr` is the address of the memory to prefetch. There are two optional arguments, `rw` and `locality`. The value of `rw` is a compile-time constant one or zero; one means that the prefetch is preparing for a write to the memory address and zero, the default, means that the prefetch is preparing for a read. The value `locality` must be a compile-time constant integer between zero and three. A value of zero means that the data has no temporal locality, so it needs not be left in the cache after the access. A value of three means that the data has a high degree of temporal locality and should be left in all levels of cache possible. Values of one and two mean, respectively, a low or moderate degree of temporal locality. The default is three.

```
for (i = 0; i < n; i ++) {
    a[i] = a[i] + b[i];
    __builtin_prefetch(&a[i+j], 1, 1);
    __builtin_prefetch(&b[i+j], 0, 1);
    /* ... */
}
```

Data prefetch does not generate faults if `addr` is invalid, but the address expression itself must be valid. For example, a prefetch of `p->next` will not fault if `p->next` is not a valid address, but evaluation will fault if `p` is not a valid address.

If the target processor does not support data prefetch, the address expression is evaluated if it includes side effects but no other code is generated and GCC does not issue a warning.