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

"I feel the need... the need for speed." – Tom Cruise

Learning Objectives

Performance optimization and cache conscious programming, 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

1. Prairielearn Assignment `Lab12a` – **this is due by the first deadline**
2. Prairielearn Assignment `Lab12b` – **this is due by the second deadline**

Guidelines

- Your goal for this lab is to transform the provided code to reduce the number of cache misses. You will see that the total execution time is also reduced.
- Both PrairieLearn Lab assignments give you **5 tries**. The assignments also give you option to download the files and develop locally. We recommend that you **download the lab files, develop locally, check correctness and achieve some speedup before submitting on PrairieLearn.**
- The **lab files are also available through Github** through the usual fetch and merge procedure.
- Our test programs will exhaustively check the output of your code. Your code should produce the same output as the one generated by the code that we have provided. Thus, you may want to enhance the test conditions in the main code to verify the correctness of your code.
- We have tested this lab on 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. If you are developing on `EWS FastX VMs`, you may see a large variation in execution time at different times because multiple user may be running jobs simultaneously. 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 scenarios) 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).
- For grading purposes, we'll be looking at the number of cache misses that your program achieves for a cache that is common in modern computers. PrairieLearn assignments will tell you the number of missies that your code achieves along with your grade.

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 ++) {
        B[i][j] = A[j][i];
    }
}
```

In this code, accesses to `B` have spatial locality as the matrix is accessed by rows; however, accesses to `A` 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 `A`.

If you consider iterations `i=0` and `i=1` you can see that the rows of `A` that were accessed during iteration `i=0` will also be accessed during iteration `i=1`. Thus, all the memory accesses to `A` 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 `A` that were brought to the cache during the execution of iteration `i=0`. However, if `A` is very large, the cache lines from `A` brought to the cache during the execution of iteration `i=0` will not fit in cache, and all the memory accesses to `A` 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 `A` 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.

`https://mediaspace.illinois.edu/media/t/1_7tua5ief`

`https://mediaspace.illinois.edu/media/t/1_2t3a1n24`

Note: Play around with the tile size to see what makes your code run the fastest. Depending on what machine you are on, you should be able to cut the execution time by as much as an order of magnitude! (Notice also that tile size is a parameter of the loop tiling transformation and is different from SIZE, which defines the size of the images.)

You can run the program using the downloadable files with the following commands:

```
make transpose
./transpose
```

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 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` prints "Image 2323". This is only the value of a *single random pixel.* This means that if you *don't* see "Image 2374" after making a transformation, you're *definitely* wrong, but if you *do* see "Image 2374", 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 saw in the first part. Incorrect outputs from the functions will cause you to lose all points for an attempt (no partial credit will be given). Make sure you test your own code with exhaustive output checks.

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 to 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! In your local development, 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. Replace the gcc `__builtin_prefetch` with the `builtin_prefetch` before submitting on PrairieLearn.

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.

Apply loop fusion technique 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

- **gcc `__builtin_prefetch` is different from the PrairieLearn `__builtin_prefetch`. Note the absence of the `__` in the name.**
- PrairieLearn will warn you if you try to use gcc `__builtin_prefetch` and stop the grading. If you somehow manage to use gcc `__builtin_prefetch` in PrairieLearn, it will have no effect on performance in PrairieLearn. You have to use the simplified version of the prefetch provided in PrairieLearn with no read/write and locality parameters.
- 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.

- You can compile the code with the command `make filter`.
- You can execute the code with the command `./filter`.

GCC support for prefetching (For local development only)

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.