CSCI 680 – Compiler Optimization for HPC
Homework 2

Due: November 9, 2016

Alexander Powell

1. **Please explain what is data locality. What kinds of data locality can we explore? Please list at least 2 loop optimizations related to data locality with examples.**

   Data locality is an important concept in compiler optimization and computer architecture. It mostly has to do with how caches are used in an architecture. In general, caches work on the assumption that data that is accesses once will probably be accessed again in the near future. This behavior is known as data locality, and it can be separated into two types. Temporal locality refers to a program reusing data that has recently been added or referenced in a cache. Spatial locality refers to a program using data that is close in location to another recently accessed location. Essentially these two types of localities come down to locality in time and in space. It should be noted that it's possible for a program to exhibit both of these locality types. Better data locality implies better performance and programs can be optimized to take advantage of data locality.

   One common optimization related to data locality is loop tiling (also known as loop blocking). The goal of loop tiling is to break a loop up into smaller pieces to get both spatial and temporal locality. This creates new inner loops so that data accessed in these inner loops can fit into the cache. However, it should be noted that this changes the iteration order, so it may not always be legal. In the example below, the code segment on the left is the original version, and on the left shows the tiles version of the same loop:

   Loop Tiling

```
for (i = 0; i < N; i++)
    for (j = 0; j < N; j++)
        y[i] += A[i][j] * x[j]
```

```
for (ii = 0; ii < N; ii+=B)
    for (jj = 0; jj < N; jj+=B)
        for (i = ii; i < ii+B; i++)
            for (j = jj; j < jj+B; j++)
                y[i] += A[i][j] * x[j]
```

   Another common optimization related to data locality is loop interchange. This transformation involves changing the order of a nested loop. This is often done to ensure that the elements of a multi-dimensional array are accessed in the order in which they are present in memory, improving spatial locality. However, it should be noted that, like loop tiling, loop interchange is not always legal because it changes the order that elements are accessed. In the example below, a loop interchange has been performed (switching the order of the i and j loops).

```
for (i = 0; i < N; i++)
    for (j = 0; j < N; j++)
        y[i] += A[i][j] * x[j]
```

```
for (j = 0; j < N; j++)
    for (i = 0; i < N; i++)
        y[i] += A[i][j] * x[j]
```

2. **Please describe what the GPU and SSE have in common and what is the difference between them.**

   SSE, or streaming SIMD extensions, is a SIMD instruction set extension to the Intel x86 architecture. SSE provides ways for programmers to manually vectorize their code to run on a CPU.

The original release of SSE only supported operations on single precision floating point numbers, but subsequent versions now provide support on both single and double precision floating point numbers, 32 and 64 bit integers, short ints, and bytes (characters). Also, 70 different instructions are now supported as well. The benefit to these streaming SIMD extensions is that is allows a data level parallelism; meaning four (or more) different operating will be carried out concurrently. This is clearly important to the performance of a program.

On the other hand, a GPU (or graphics processing unit), is a specialized circuit designed to rapidly manipulate and alter memory to accelerate the creation of images in a frame buffer intended for output to a display. GPUs are commonly used in embedded systems, mobile phones, personal computers, and gaming consoles. However, as noted above, SSE cannot be used on a GPU, only CPUs. However, GPUs (especially GPGPUs) are generally suited to computations that exhibit data level parallelism to exploit the wide vector width SIMD architecture of the GPU.

3. **What is Super Word Level Parallelism (SLP)? What is the difference between SLP and traditional data parallelism?**

Superword level parallelism (SLP) is a vectorization technique based on loop unrolling and basic block vectorization. It is distinct from loop vectorization algorithms in that it can exploit parallelism of inline code, such as manipulating coordinates, color channels or in loops unrolled by hand. It is available in the GCC compiler since version 4.3 and is also supported by LLVM. The concept was introduced by Samuel Larsen and Saman Amarasinghe from MIT labs. They introduced it as a novel way of viewing parallelism in multimedia and scientific applications.

Most traditional vectorization techniques involve extracting inter-iteration (one iteration to another) parallelism from a program. However, superword level parallelism deals with exploiting intra-iteration (within the same iteration) parallelism from a program. The term *superword* denotes wide-length data packed for intra-iteration parallelism whereas *vector* refers to inter-iteration parallelism.

4. **What is data dependency? How many types of data dependencies do we have? What are they? What kinds of problems could they cause? Give some concrete examples.**

A data dependency is a situation in which a program statement refers to the data of a preceding statement. The three main types of data dependencies are flow, anti, and output dependencies. Flow dependencies are also known as read after write dependencies. Data dependencies often cause problems when attempting to optimize code in some way by transforming it. For example, in the loop interchange and loop tiling examples given in problem 1, no data dependencies existed so our transformations were valid. However, in cased where data dependencies do exist, more work is required to properly optimize the statements.

An example of a flow dependency is shown below. Flow dependencies occur when an instruction depends on the result of a previous instruction before it can execute.

```
A = 3
B = A
C = B
```

In this case, the instruction B = A must execute before C = B, or the program will not output the correct results.

The next type of data dependency is an anti-dependency, also called a write after read dependency. This kind of dependency occurs when an instruction requires a value that is later updated. For example, the order of the following code cannot be changed or executed in parallel without changing the outcome of the program.

```
B = 3
A = B + 1
B = 7
```

The third class of dependency is an output dependency, or write after write dependency. This occurs when the ordering of the instructions can affect the final output value of a variable. For example, in the code segment shown above there is an output dependency associated with instructions 1 and 3, because depending on the order of execution the final value could be different. So, in summary, there are three main types of dependencies: read-after-write (RAW), write-after-write (WAW), and write-after-read (WAR).

5. **What are the main challenges of mapping irregular applications to SIMD architecture? Please list at least 3 with examples.**

There are a number of problems associated with irregular mappings on SIMD architectures. The three most commons are:

1) Non-contiguous memory references
2) Arbitrarily misaligned memory references
3) Dependencies (especially pertaining to loops)

Non-contiguous memory references (also known as disjoint memory references) refer to how the memory is allocated by a program. A contiguous block of memory implies that a number of bytes of storage are allocated in one big chunk, as opposed to being split into a number of smaller chunks. Due to the nature of indirect array accesses, it is often the case that these programs experience disjoint memory references.

Data alignment refers to storing data at a memory address equal to some multiple of the word size. This can increase the system's performance due to the way the CPU handles memory. For example, depending on whether data is aligned or not in the address space, this can increase or decrease the number of load and store instructions that may need to be performed, both of which have longer latencies than other operations like ALU instructions. As an example, consider the case where a computer's word size is 4 bytes, so data should be read at a memory address which is some multiple of 4. A load word instruction will fetch the word in memory with the given starting address. However, if an unaligned word is required, stored at location 14 for instance, two loads will have to be performed: one at location 12, and another at 16. Then some sort of calculation has to be performed to get the requested word. Clearly data alignment is important.

Finally, loop carried dependencies pose a big problem to irregular mapping applications. Due to the array indirection, it's not often evident at compile time whether a dependency exists, so parallelization is difficult. However, while it may not be possible to fully vectorize a loop with array indirection, certain transformations are possible to vectorize components of the loop. For example, loop fission, loop distribution, and reduction processing can be used to improve the performance where possible.