CMPSCI 630 Systems **Spring 2022**

Lecture 2

Lecturer: Emery Berger *Scribe: Shira Epstein, Peter Tran*

2.1 Is FORTRAN Still in Use?

We are continuing from last lecture, this time looking at reasons why FORTRAN is still in use.

The truth is that most of us are actually FORTRAN clients because of the fact that we use numpy, which is implemented in FORTRAN. But why is numpy implemented in FORTRAN? Some reasons as given by the class: legacy, optimized, low level, and familiarity. Perhaps they are valid reasons, but the most important reason is that numpy uses BLAS (Basic Linear Algebraic Subprograms). And BLAS happens to be written in FORTRAN.

(library) BLAS

————— (language)

Hardware

A final note to discuss before moving on, there is a program called ”f2c” that can take a FORTRAN program and convert it into a C program. The downside being that the converted C program ends up running slower than the FORTRAN program.

2.2 Threading and SIMD

Now lets talk about the programming language C. C is considered a ”Systems” language because C takes addresses of variables. Thus, C deals with pointers which we will see soon is problematic.

We are also aware that C has threads! Threads are a useful tool in ”unrolling” big loops into smaller chunks, simply because cache memory is limited. Similar to the divide and conquer strategy where we break the problem into more manageable pieces. It is important, however, for the programmer when unrolling loops to be mindful of cache size. The programmer must also observe granularity, because making a thread is costly. The OS is deeply involved in thread scheduling, and talking to the OS is slow.

There is a concept similar to threads that is called vector instructions. What are vector instruction? They are instructions that operate on whole arrays without loops. An analogy to that is scaling a matrix by a scalar in parallel. There is a type of parallel processing called ”SIMD” (single instruction multiple data) that performs the same operation on multiple data sets concurrently. Where each thread may be doing a different operation, in the SIMD philosophy there is only one instruction, to be executed over a large data set. So SIMD and threads are similar but distinct.

It should be noted that there is a software called OpenMP that allows FORTRAN to support threading. And it actually works wonderfully.

2.3 Why is FORTRAN Efficient?

The question we can ask is if C can do the above, why is FORTRAN more efficient? For one, variables in C are disjoint so the compiler has to use pointer analysis to check pointer validity. This can be quite costly. Another concept that we are familiar with is something called ”race condition”. That is the threads are executing independently and the order in which they finish introduces bugs/problems to the solution. This leads to non-determinism because we are not completely sure our solution will be the same each run. Finally there is a term called ”path explosion”. This is where pointers are involved with complex recursive statements. Not only does the number of paths increase exponentially with large recursive statements, but the compiler is overwhelmed at the amount of pointer analysis to be done.

A note about C and pointer analysis, it should be known that C does something called ”Heroic Pointer Analysis”. It is a best effort attempt to validate pointer addresses. Pointer analysis can be path sensitive, flow sensitive, or context sensitive.

Back to FORTRAN, we note another advantage that this language has in the department of efficiency. FORTRAN uses arrays for everything. We can pass in an array and it is a guarantee that this is just a chunk of memory. The variable memories do not overlap. So we avoid the pointer madness.

2.4 Branch Predictions

What are branch predictions? Branch prediction is a method that a compiler uses to increase efficiency for long loops. Clang is a modern C/C++ compiler that does this. Essentially the compiler remembers the previous results of the initial iterations through the loop, and makes ”predictions” about the results of the subsequent iterations. It then performs a check for correctness at the end taking advantage of locality to save time.

branch – speculation – results – check results

is faster than

branch – check results – results

Object splitting is a layout optimization. It allows all of our information to be in close by cache lines. However, a relatively recent result says that we cannot make anything near an optimal layout optimization, not even a polynomial approximation. A super polynomial in the worst case. It is all heuristics except you have very regular programs.

2.5 How does C and Java Differ in Memory Storage

Let’s say we wanted to create a 256 by 256 array. Like so:

float m[256][256]

How does this look like internally?

In C :

256 x 256 chunk
...
...

We are given a chunk of memory, that is ready to be accessed.

In Java (figure below is an example of a "ragged array")

pointer →	size 256 list
pointer →	size 256 list
pointer →	...
pointer →	...

Java has smaller chunks of memory, with pointers to objects. Therefore Java has a problem with locality, because it must follow the pointers to access data. In other words, every single time you access something you have to go through a pointer indirection. This takes time. And the structure of the internal storage may resembles "ragged arrays", which destroys some memory locality optimizations.

Java is also an in time compilation language. It does object inlining (taking objects and combining them) on the fly. It also does optimizations on the fly, addressing issues at the garbage collection layer

Thus it can be said that for the same program, C runs much more faster than Java.