CS 380C: Advanced Topics in Compilers
Administration

• **Instructor: Keshav Pingali**
  – Professor (CS, ICES)
  – ACES 4.126A
  – pingali@cs.utexas.edu

• **TA: Oswaldo Olivo**
  – Graduate student (CS)
  – olivo@cs.utexas.edu
Meeting times

• Lecture:
  – TTh 9:30-11:00AM, RLM 6.126

• Office hours:
  – Keshav Pingali: Tuesday 11-12 AM, ACES 4.126
Prerequisites

• **Compilers and architecture**
  – Some background in compilers (front-end stuff)
  – Basic computer architecture

• **Software and math maturity**
  – Able to implement large programs in C/C++
  – Comfortable with abstractions like graph theory

• **Ability to read research papers and understand content**
Course material

• Website for course

• All lecture notes, announcements, papers, assignments, etc. will be posted there

• No assigned book for the course
  – but we will put papers and other material on the website as appropriate
Coursework

• 4-5 programming assignments and problem sets
  – Work in pairs

• Term project
  – Substantial implementation project
  – Based on our ideas or yours in the area of compilers
  – Work in pairs

• Paper presentations
  – Towards the end of the semester
What do compilers do?

- **Conventional view of compilers**
  - Program that analyzes and translates a high-level language program automatically into low-level machine code that can be executed by the hardware
  - May do simple (scalar) optimizations to reduce the number of operations
  - Ignore data structures for the most part
- **Modern view of compilers**
  - Program for translation, transformation and verification of high-level language programs
  - Reordering (restructuring) the computations is as important if not more important than reducing the amount of computation
  - Optimization of data structure computations is critical
  - Program analysis techniques can be useful for other applications such as
    - debugging,
    - verifying the correctness of a program against a specification,
    - detecting malware, …. 
Why do we need translators?

- **Bridge the “semantic gap”**
  - Programmers prefer to write programs at a high level of abstraction
  - Modern architectures are very complex, so to get good performance, we have to worry about a lot of low-level details
  - Compilers let programmers write high-level programs and still get good performance on complex machine architectures

- **Application portability**
  - When a new ISA or architecture comes out, you only need to reimplement the compiler on that machine
  - Application programs should run without (substantial) modification
  - Saves a huge amount of programming effort
Complexity of modern architectures: AMD Barcelona Quad-core Processor
Discussion

• To get good performance on modern processors, program must exploit
  – coarse-grain (multicore) parallelism
  – memory hierarchy (L1,L2,L3,..)
  – instruction-level parallelism (ILP)
  – registers
  – ….

• Key questions:
  – How important is it to exploit these hardware features?
    • If you have n cores and you run on only one, you get at most 1/n of peak performance, so this is obvious
    • How about other hardware features?
  – If it is important, how hard is it to do this by hand?

• Let us look at memory hierarchies to get a feel for this
  – Typical latencies
    • L1 cache: ~ 1 cycle
    • L2 cache: ~ 10 cycles
    • Memory: ~ 500-1000 cycles
Software problem

• Caches are useful only if programs have locality of reference
  – temporal locality: program references to given memory address are clustered together in time
  – spatial locality: program references clustered in address space are clustered in time

• Problem:
  – Programs obtained by expressing most algorithms in the straight-forward way do not have much locality of reference
  – Worrying about locality when coding algorithms complicates the software process enormously.
Example: matrix multiplication

```plaintext
DO I = 1, N  //assume arrays stored in row-major order
  DO J = 1, N
    DO K = 1, N
      C(I,J) = C(I,J) + A(I,K)*B(K,J)
    END DO K
  END DO J
END DO I
```

- Great algorithmic data reuse: each array element is touched $O(N)$ times!
- All six loop permutations are computationally equivalent (even modulo round-off error).
- However, execution times of the six versions can be very different if machine has a cache.
DO I = 1, N
  DO J = 1, N
    DO K = 1, N
      C(I,J) = C(I,J) + A(I,K)*B(K,J)

• Large cache scenario:
  – Matrices are small enough to fit into cache
  – Only cold misses, no capacity misses
  – Miss ratio:
    • Data size = 3 N²
    • Each miss brings in b floating-point numbers
    • Miss ratio = \(\frac{3 N^2}{b \times 4N^3} = \frac{0.75}{bN} = 0.019\) (b = 4, N=10)
DO I = 1, N
DO J = 1, N
  DO K = 1, N
    C(I,J) = C(I,J) + A(I,K)*B(K,J)

• Small cache scenario:
  – Matrices are large compared to cache/row-major storage
  – Cold and capacity misses
  – Miss ratio:
    • C: \(N^2/b\) misses (good temporal locality)
    • A: \(N^3/b\) misses (good spatial locality)
    • B: \(N^3\) misses (poor temporal and spatial locality)
    • Miss ratio \(\Rightarrow 0.25 \frac{(b+1)}{b} = 0.3125\) (for \(b = 4\))
MMM Experiments

- Simulated L1 Cache Miss Ratio for Intel Pentium III
  - MMM with $N = 1…1300$
  - 16KB 32B/Block 4-way 8-byte elements
Quantifying performance differences

DO I = 1, N    //assume arrays stored in row-major order
    DO J = 1, N
        DO K = 1, N
            C(I,J) = C(I,J) + A(I,K)*B(K,J)

• Typical cache parameters:
  – L2 cache hit: 10 cycles, cache miss 70 cycles
• Time to execute IKJ version:
  \[2N^3 + 70 \times 0.13 \times 4N^3 + 10 \times 0.87 \times 4N^3 = 73.2 N^3\]
• Time to execute JKI version:
  \[2N^3 + 70 \times 0.5 \times 4N^3 + 10 \times 0.5 \times 4N^3 = 162 N^3\]
• Speed-up = 2.2
• Key transformation: loop permutation
Even better.....

- Break MMM into a bunch of smaller MMMs so that large cache model is true for each small MMM
  - large cache model is valid for entire computation
  - miss ratio will be $0.75/bt$ for entire computation where $t$ is
Loop tiling/blocking

- Break big MMM into sequence of smaller MMMs where each smaller MMM multiplies sub-matrices of size $txt$.
- Parameter $t$ (tile size) must be chosen carefully
  - as large as possible
  - working set of small matrix multiplication must fit in cache
Speed-up from tiling/blocking

- Miss ratio for block computation
  = miss ratio for large cache model
  = 0.75/bt
  = 0.001 (b = 4, t = 200)

- Time to execute tiled version =
  \[2N^3 + 70 \times 0.001 \times 4N^3 + 10 \times 0.999 \times 4N^3 = 42.3N^3\]

- Speed-up over JKI version = 4
Observations

• Locality optimized code is more complex than high-level algorithm.
• Locality optimization changed the order in which operations were done, not the number of operations
• “Fine-grain” view of data structures (arrays) is critical
• Loop orders and tile size must be chosen carefully
  – cache size is key parameter
  – associativity matters
• Actual code is even more complex: must optimize for processor resources
  – registers: register tiling
  – pipeline: loop unrolling
  – Optimized MMM code can be ~1000’s of lines of C code
• Wouldn’t it be nice to have all this be done automatically by a compiler?
  – Actually, it is done automatically nowadays…
Performance of MMM code produced by Intel’s Itanium compiler (-O3)

Goto BLAS obtains close to 99% of peak, so compiler is pretty good!

GFLOPS relative to -O2; bigger is better

92% of Peak Performance

Goto BLAS obtains close to 99% of peak, so compiler is pretty good!
Discussion

• Exploiting parallelism, memory hierarchies etc. is very important
• If program uses only one core out of n cores in processors, you get at most 1/n of peak performance
• Memory hierarchy optimizations are very important
  – can improve performance by factor of 10 or more
• Key points:
  – need to focus on data structure manipulation
  – reorganization of computations and data structure layout are key
  – few opportunities usually to reduce the number of computations
Organization of modern compiler

Source program

Front-end

Abstract syntax tree

High-level Optimizer

Low-level representation (3-address code,....)

Low-level Optimizer

Augmented low-level representation

Code generator

Assembly or machine code

Our focus
Front-end

• **Goal:** convert linear representation of program to hierarchical representation
  – Input: text file
  – Output: abstract syntax tree + symbol table

• **Key modules:**
  – Lexical analyzer: converts sequence of characters in text file into sequence of tokens
  – Parser: converts sequence of tokens into abstract syntax tree + symbol table
  – Semantic checker: (eg) perform type checking
High-level optimizer

• Goal: perform high-level analysis and optimization of program
• Input: AST + symbol table from front-end
• Output: Low-level program representation such as 3-address code
• Tasks:
  – Procedure/method inlining
  – Array-pointer dependence analysis
  – Loop transformations: unrolling, permutation, tiling, jamming, ….
Low-level optimizer

- Goal: perform scalar optimizations on low-level representation of program
- Input: low-level representation of program such as 3-address code
- Output: optimized low-level representation + additional information such as def-use chains
- Tasks:
  - Dataflow analysis: live variables, reaching definitions, ...
  - Scalar optimizations: constant propagation, partial redundancy elimination, strength reduction, …
Code generator

- Goal: produce assembly/machine code from optimized low-level representation of program
- Input: optimized low-level representation of program from low-level optimizer
- Output: assembly/machine code for real or virtual machine
- Tasks:
  - Register allocation
  - Instruction selection
Discussion (I)

• Traditionally, all phases of compilation were completed before program was executed

• New twist: virtual machines
  – Offline compiler:
    • Generates code for virtual machine like JVM
  – Just-in-time compiler:
    • Generates code for real machine from VM code while program is executing

• Advantages:
  – Portability
  – JIT compiler can perform optimizations for particular input
Discussion (II)

• On current processors, accessing memory to fetch operands for a computation takes much longer than performing the computation
  \[ \Rightarrow \] performance of most programs is limited by memory latency rather than by speed of computation (memory wall problem)
  \[ \Rightarrow \] reducing memory traffic (locality) is more important than optimizing scalar computations

• Another problem: power
  - takes much more power to move data than to perform an arithmetic operation
  - exploiting locality is critical for power management as well
Course content (scalar stuff)

- **Introduction**
  - compiler structure, architecture and compilation, sources of improvement
- **Control flow analysis**
  - basic blocks & loops, dominators, postdominators, control dependence
- **Data flow analysis**
  - lattice theory, iterative frameworks, reaching definitions, liveness
- **Static-single assignment**
  - static-single assignment, constant propagation.
- **Global optimizations**
  - loop invariant code motion, common subexpression elimination, strength reduction.
- **Register allocation**
  - coloring, allocation, live range splitting.
- **Instruction scheduling**
  - pipelined and VLIW architectures, list scheduling.
Course content (data structure stuff)

- **Array dependence analysis**
  - integer linear programming, dependence abstractions.

- **Loop transformations**
  - linear loop transformations, loop fusion/fission, enhancing parallelism and locality

- **Self-optimizing programs**
  - empirical search, ATLAS, FFTW

- **Analysis of pointer-based programs**
  - points-to and shape analysis

- **Parallelizing graph programs**
  - amorphous data parallelism, exploiting amorphous data-parallelism

- **Program verification**
  - Floyd-Hoare style proofs, model checking, theorem provers
Lecture schedule

• See

• Some lectures will be given by guest lecturers from my group and from industry