Advanced Compilation Topics

Just-In-Time (JIT) Compilation
- **Approach**
  - Compile portions of code (or entire program) at run time
  - Compilation time becomes execution time
- **Combination of compiler & interpreter**
  - Achieve speed of compiled code
  - Take advantage of benefits of interpreter
- **Advantages**
  - Can target specific architecture (e.g., Xeon vs x86)
  - Can perform optimizations on dynamically linked library code
  - Can run compiler in sandbox for safety
- **Java extensively uses JIT**
  - Transforms Java source into Java byte code
  - Byte code is more compact than source code
  - JIT compilation on byte code is simpler / faster than on Java source

Run-time Optimization
- **Approach**
  - Compiler identifies important conditions in code
  - Compiler then generates for each condition
    - Run time test for condition
    - Multiple versions of code optimized for different condition, or optimized code that can be parameterized at run time
  - Program dynamically selects version (or changes parameters) at run time
- **Advantages**
  - Resulting programs can adapt at run time to a variety of conditions
  - Approach can be extended to “compiling” binary files
    - Applying run-time optimization to binaries directly w/o source

Profile-based Compilation
- **Approach**
  - Compiler inserts instrumentation in program
  - Run instrumented program on representative input
  - Recompile using profile to guide optimization
  - Optimized code
- **Advantages**
  - Can obtain more information than static analysis
  - Identify frequently executed code
  - Identify likely result of conditional branches
  - Estimate loop iterations
  - No run time overhead for final optimized code
- **Disadvantages**
  - Longer compilation / optimization process
  - Actual application behavior may not match representative input

Graphics Processing Units (GPUs)
- **GPUs**
  - Few instructions lots of data
  - Branch prediction
  - Reuse and locality
  - Needs OS
  - Complex sync
  - Latency machines
- **CPUs**
  - Many instructions little data
  - Out of order exec
  - Branch prediction
  - Little reuse
  - Data parallel
  - No OS
  - Simple sync
  - Throughput machines
Compiling for GPUs

- **Approach**
  - GPUs can compute vector / stream operations in parallel
  - Using special libraries (e.g., CUDA) to copy / process data
  - Requires programs for both GPU & CPU
  - Compiler can simplify process of generating GPU code
  - PGI compiler relies on user-inserted annotations to specify parallel region, vector operations

- **Advantages**
  - Supercomputer-like FP performance on commodity processors

- **Disadvantages**
  - Performance tuning difficult
  - Large speed gap between compiler-generated and hand-tuned code

Compiling for GPUs - Matrix Multiplication Example

- **Annotated Fortran for PGI compiler (compiled to CUDA)**
  ```fortran
  !$acc region
  !$acc do parallel
do j=1,m
do k=1,p
   !$acc do parallel, vector(2)
do i=1,n
    a(i,j) = a(i,j) + b(k,i)*c(k,j)
   enddo
  enddo
  !$acc end region
  ```

Compiling for GPUs - Matrix Multiplication Example

- **Hand-written GPU code using CUDA**
  ```c
  __global__ void
  matmulfloat(float* A, float* B, float* C, int m, int n2, int N3)
  {
   int bx = blockDim.x, by = blockDim.y;
   int tx = threadIdx.x, ty = threadIdx.y;
   __syncthreads();
   C[tx*N3 + ty] = C[tx*N3 + ty] + B[ty*N3+tx];
   __syncthreads();
   C[tx*N3 + ty] = C[tx*N3 + ty] + B[ty*N3+tx];
  }
  ```

Power-Aware Computing

- **Cooking an egg on the CPU**
Power Issues in Microprocessors

Capacitive (Dynamic) Power

Static (Leakage) Power

Di/Dt (Vdd/Gnd Bounce)

Temperature

Code Optimizations for Low Power

- Most code optimizations reduce power use
  - Code executing fewer instructions use less power
- Other optimizations affect power, not performance
  - Reorder instructions
    - Reduce switching effect of functional units and I/O buses
  - Operand swapping
    - Swap the operands at the input of multiplier
    - Result is unaltered, but power changes significantly!
  - Use processor-specific instruction styles
    - On ARM the default int type is ~ 20% more efficient than char or short (sign/zero extension)
  - Use processor-specific features
    - Shut off unused registers to save power
    - Reduce voltage to save power when performance not needed