Advanced Compilation Topics

Some slides thanks to Prof. K. Hazelwood @ UVA

Advanced topics

- **Static Compilation**
  → All work is done ahead of time
- **Just-in-Time Compilation**
  → Compile parts of program at run time
- **Profile-based optimization**
  → Use run-time profile to guide optimization
- **Run-time optimization**
  → Include multiple options in binary
- **Compiling for GPUs**
  → Handle specialized libraries & algorithms
- **Compiling for low power**
  → Reducing power usage
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 soapbox 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

Profile-based Compilation

- **Approach**
  - Compiler inserts instrumentation in program
  - Run instrumented program on representative input → profile
  - 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
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

Graphics Processing Units (GPUs)

• CPUs
  • Lots of instructions little data
    ▪ Out of order exec
    ▪ Branch prediction
  • Reuse and locality
  • Task parallel
  • Needs OS
  • Complex sync
  • Latency machines

• GPUs
  • Few instructions lots of data
    ▪ SIMD
    ▪ Hardware threading
  • 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 CPU & GPU
  → 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

• Original Fortran
  do i = 1,n
      do j = 1,m
          do k = 1,p
              a(i,j) = a(i,j) + b(i,k)*c(k,j)
          enddo
      enddo
  enddo
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
    !$acc do parallel, vector(2)
do i=1,n
  a(i,j) = a(i,j) + b(i,k)*c(k,j)
enddo
enddo
enddo
!$acc end region
```

Compiling for GPUs – Matrix Multiplication Example

• Hand-written GPU code using CUDA

```c
__global__ void matmulKernel( float* C, float* A, float* B, int N2, int N3 ){
  int bx = blockIdx.x, by = blockIdx.y;
  int tx = threadIdx.x, ty = threadIdx.y;
  int aFirst = 16 * by * N2;
  int bFirst = 16 * tx;
  float Csub = 0;

  for( int j = 0; j < N2; j += 16 ) {
    __shared__ float Atile[16][16], Btile[16][16];
    Atile[ty][tx] = A[aFirst + j*N2 + ty + tx];
    Btile[ty][tx] = B[bFirst + j*N3 + b + N3 * ty + tx];
    __syncthreads();

    for( int k = 0; k < 16; ++k )
      Csub += Atile[ty][k] * Btile[k][tx];
    __syncthreads();
  }

  int c = N3 * 16 * bx + 16 * by;
  C[c + N3 * ty + tx] = Csub;
}
```
Compiling for GPUs – Matrix Multiplication Example

- **Hand-written CPU code using CUDA**

```c
void matmul( float* A, float* B, float* C, 
    size_t N1, size_t N2, size_t N3 )
{
    void *devA, *devB, *devC;
    cudaSetDevice(0);

cudaMalloc( &devA, N1*N2*sizeof(float) );
cudaMalloc( &devB, N2*N3*sizeof(float) );
cudaMalloc( &devC, N1*N3*sizeof(float) );

cudaMemcpy( devA, A, N1*N2*sizeof(float), cudaMemcpyHostToDevice );
cudaMemcpy( devB, B, N2*N3*sizeof(float), cudaMemcpyHostToDevice );

dim3 threads( 16, 16 );
dim3 grid( N1 / threads.x, N3 / threads.y);

matmulKernel<<< grid, threads >>>( devC, devA, devB, N2, N3 );

cudaMemcpy( C, devC, N1*N3*sizeof(float), cudaMemcpyDeviceToHost );
cudaFree( devA );
cudaFree( devB );
cudaFree( devC );
}
```

Power-Aware Computing

![Cooking an egg on the CPU](image_url)
Power Issues in Microprocessors

Capacitive (Dynamic) Power

\[
\text{Dynamic Power} = \frac{1}{2} CV^2 f
\]

- Dynamic Energy (when switching) is proportional to Capacitance \( \times \) Voltage\(^2\)
- Since pulse is 0 → 1 → 0 or 1 → 0 → 1,
  Energy of a single transition is proportional to \( \frac{1}{2} \times \text{Capacitance} \times \text{Voltage}^2 \)
- Power is just energy per transition times frequency of transitions: proportional to \( \frac{1}{2} \times \text{Capacitance} \times \text{Voltage}^2 \times \text{Frequency} \)
- To reduce power used
  \( \rightarrow \) Reduce frequency (2 GHz → 1 GHz = 50% less power)
  \( \rightarrow \) Reduce voltage (1.5V → 1.4V = 15% less power)
  \( \rightarrow \) Reduce bit-level value changes
  (01 → 10 vs. 00 → 01 = 50% fewer bits changed)
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 at 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