Tricking Compilers to do work for you

Fernanda Foertter
Oak Ridge Leadership Computing Facility (OLCF)
WHAT IS A COMPILER?
WHAT IS A COMPILER?

High Level

MAGIC

Machine
CODE OPTIMIZER

• Reads what you've written, determines what can be optimized

• Writes intermediate code, analyses, repeats

• **The more obvious your intentions, the better**

Control Flow

- branching
- loops
- routines

Data Flow

- de/allocs
- data reuse
- cache ops
FUNCTION INLINING

• Adds pieces of a small subroutine or function within loops

• saves call overhead

```c
for (int i = 0; i < 100; i++)
{
    a[i] = square_area_function(length[i]);
}

for (int i = 0; i < 100; i++)
{
    a[i] = length[i]^2;
}
```
LOOP UNROLLING

• removes loop tests and loop control

• make sure your loop length is obvious

• avoid sudden breaks

```c
for (int i = 0; i < 50; i++)
{
    calculate_function(i);
}
```

```c
for (int i = 0; i < 50; i+=5)
{
    calculate_function(i);
    calculate_function(i+1);
    calculate_function(i+2);
    calculate_function(i+3);
    calculate_function(i+4);
}
```
INSTRUCTION OPTIMIZATION

• Making use of built-in instructions or specialized hardware.

• Inlining small routines/function calls

• moving code out of loops that are unnecessary

```c
for (int i = 0; i < n; i++) {
    x = a + b;
    x2 = x*x;
    for (int i = 0; i < n; i++) {
        y[i] = x2 + 3*i;
    }
}
```
DATA-FLOW ANALYSES

• depending on the flow or path, data may need to be reused. e.g.: different variables point to the same location, or shuffle instructions whose vars don’t depend on each other but avoids extraneous movement

• remove dead store e.g.: a var never gets used but gets repeatedly computed
BRANCHING

- e.g.: if statement: compiler tries to guess which way, sometimes wasting extra cycles

If A (do something)  
\[ \text{it's A} \]  
else (do otherthing)
Begin loop 10 times
read instruction and decode
load \( y[0] \)
load \( x[0] \)
add them
save result
End loop

<table>
<thead>
<tr>
<th>( y[ ] )</th>
<th>( x[ ] )</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
</tr>
<tr>
<td>7</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>7</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>Flag</td>
<td>Description</td>
</tr>
<tr>
<td>------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>none</td>
<td>Optimizes for speed; no flags, approx -O2 level</td>
</tr>
<tr>
<td>-fast</td>
<td>Maximizes speed across the entire program. Aggressive.  Includes -ipo -O3 -no-prec-div -static -xHost</td>
</tr>
<tr>
<td>-fast -no-ipo</td>
<td>No IPO</td>
</tr>
<tr>
<td>-O0</td>
<td>Disables optimizations</td>
</tr>
<tr>
<td>-O1</td>
<td>Optimizes for speed but keeps binary small</td>
</tr>
<tr>
<td>-O2</td>
<td>Speed + vectorization, inlining</td>
</tr>
<tr>
<td>-O3</td>
<td>O2 + aggressive loop transformations</td>
</tr>
<tr>
<td>-mkl=cluster</td>
<td>Single threaded MKL (BLAS/LAPACK/scALAPACK/FFTW)</td>
</tr>
<tr>
<td>-mkl=parallel</td>
<td>Multi-Threaded MKL (No ScALAPACK)</td>
</tr>
<tr>
<td>-xAVX</td>
<td>Default on Eos</td>
</tr>
</tbody>
</table>
WHICH COMPILER IS IMPORTANT

• They don't all cover the same parts of the standard

• Some do better if they also manufacture the hardware

• Implementations vary!

• Some are pedantic

• Some leaving you hanging

• Some optimize better than others on certain parts
<table>
<thead>
<tr>
<th></th>
<th>Titan Nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Node</strong></td>
<td>AMD Opteron 6200 Interlagos (16 cores)</td>
</tr>
<tr>
<td></td>
<td>2.2 GHz</td>
</tr>
<tr>
<td></td>
<td>32 GB (DDR3)</td>
</tr>
<tr>
<td><strong>Accelerator</strong></td>
<td>Tesla K20x</td>
</tr>
<tr>
<td></td>
<td>(2688 CUDA cores)</td>
</tr>
<tr>
<td></td>
<td>732 MHz</td>
</tr>
<tr>
<td></td>
<td>6 GB (DDR5)</td>
</tr>
<tr>
<td><strong>Network</strong></td>
<td>Gemini High Speed Interconnect</td>
</tr>
<tr>
<td></td>
<td>3D Torus</td>
</tr>
<tr>
<td><strong>Storage</strong></td>
<td>Luster Filesystem</td>
</tr>
<tr>
<td></td>
<td>5 PB</td>
</tr>
<tr>
<td><strong>Archive</strong></td>
<td>High-Performance Storage System (HPSS)</td>
</tr>
<tr>
<td></td>
<td>29 PB</td>
</tr>
</tbody>
</table>
ARCHITECTURE IS IMPORTANT

• Cache size

• Special instructions

• Special functions
XC-30
INTEL

- Dual OPI SMP Links
- 4 Channels DDR3
- Intra Cabinet/Blue (Optic Cable x 10 links) 12.5 Gbps
- Chassis/Green (Backplane x 15 links) 14 Gbps
- PCIe-3 16 bits at 8.0 GT/s per direction
- Aries 48-port Router 4 NICs, 2 router tiles each 30 router tiles for interconnect
- On-Die Interconnect
- Higher BW PCIe Gen3
- Higher Socket to Socket BW
- Significantly Higher Cache BW
- 4 channels of DDR3

20MB Cache
ACCELERATORS
HYBRID PROGRAMMING MODEL

• On Jaguar, with 299,008 cores, we were seeing the limits of a single level of MPI scaling for most applications

• To take advantage of the vastly larger parallelism in Titan, users need to use hierarchical parallelism in their codes
  • Distributed memory: MPI, SHMEM, PGAS
  • Node Local: OpenMP, Pthreads, local MPI communicators
  • Within threads: Vector constructs on GPU, libraries, OpenACC

• These are the same types of constructs needed on all multi-PFLOPS computers to scale to the full size of the systems!
HYBRID PROGRAMMING MODEL
LOOP PARALLELIZATION

Loop
i=1,40
cpu 1

Loop
i=1,10
cpu 1

Loop
i=11,20
cpu 2

Loop
i=21,30
cpu 3

Loop
i=31,40
cpu 4