Performance Analysis and Optimization
MAQAO Tool

Andrés S. CHARIF-RUBIAL
Emmanuel OSERET
{achar,emmanuel.oseret}@exascale-computing.eu
Exascale Computing Research
11th VI-HPS Tuning Workshop
Outline

- What’s new compared to TW10
  - STAN (ivy bridge), Profiler, Memory characterization
- Introduction
- MAQAO Tool
- Static Analysis
- Dynamic Analysis
- Building performance evaluation tools
- Conclusion
Introduction

Performance analysis

- Understand the performance of an application
  - How well it behaves on a given machine

- What are the issues?

- Generally a multifaceted problem
  - Maximizing the number of views = better understand

- Use techniques and tools to understand issues

- Once understood ➔ Optimize application
Outline

- Introduction
- MAQAO Tool
- Static Analysis
- Dynamic Analysis
- Building performance evaluation tools
- Conclusion
MAQAO Tool

Components

- Framework: a modular approach
  - Expose MAQAO internals
  - Scripting Language (Lua)
  - User defined plugins

- Tool
  - Built on top of the Framework
  - Loop-centric approach
  - Produce reports
  - Methodology: Top/Down iterative
MAQAO Tool
Framework overview

Binary Manipulation Layer
- Disassembler
- Generator
- Disassemble
- Re-assemble
- Patch/Rewrite

Code Structuration and Analysis Layer
- Functions
- Loops
- Instructions
- Basic blocks
- Demangling
- Debug symbols
- Other analysis algorithms (SSA, Dominance, ...)

Lua Plugins

API bindings to low-level layers
- STAN
- MTL
- MIL
- Profiler
.framework overview

- Hierarchy of objects

- Project
- Binaries AND/OR Libraries
- Functions
- Loops
- Basic blocks
- Instructions
Scripting language
- Lua language: simplicity and productivity
- Fast prototyping
- Lua API: Access to
  - the code structuration and analysis layer
  - the binary rewriting layer
  - already existing modules
- Customized static analysis
- Customized dynamic analysis
Example of script: Display memory instructions

```plaintext
// Create a project and load a given binary
local project = project.new("targeting load memory instructions");
local bin = proj:load (arg[1], 0);
// Go through the abstract objects hierarchy and filter only load memory instructions
for f in bin:functions() do
    for l in f:innermost_loops() do
        for b in l:blocks() do
            for i in b:instructions() do
                if (i:is_load()) then
                    local memory_operand = i:get_first_mem_oprnd();
                    print(i);
                    print(memory_operand);
            end
        end
    end
end
```
MAQAO Tool
Components

- Framework: a modular approach
  - Expose MAQAO internals
  - Scripting Language (Lua)
  - User defined plugins

- Tool
  - Built on top of the Framework
  - Loop-centric approach
  - Produce reports
  - Methodology: Top/Down iterative
MAQAO Tool
Iterative workflow

Source code correlation

Analyzer

Loop-centric view

Loop detection

Code structuration

CFG
CG
DDG

Disassemble

Binary code

Compiler

Modify
Optimize

Analyses

Dynamic

Static

Reports

End User
Developer

Source code

Runtime

External Developers

External Developers => New modules

Analyses

Andrés S CHARIF-RUBIAL

MAQAO Tool

11
MAQAO Tool

Methodology

- Decision tree: smallest possible
- Detect hot spots (functions, loops)
- Specialized modules
  - Memory behavior characterization (MTL)
  - Code quality assessor (STAN)
- Focus on specific problems: value profiling
  - Specialization, Memoization, Instance based VP
  - Ongoing effort
Iterative approach:
- User chooses to start over again if it is worth
- Based on reports, modify source code

Exploit compiler to the maximum
- Inlining
- Flags
- Optimization levels
- Pragmas: unroll, vectorize
- Intrinsics
- Structured code (compiler sensitive)
Static analysis

**STAN module: code quality assesor**

- Loop-centric
- Performance modeling (execution pipeline)
  - Predict performance on one core
- Take into account target (micro)architecture
- Assess code quality
  - Metrics
  - Propose solutions
Static analysis

**STAN module: code quality assessor**

- Simulates the target micro-architecture
  - Instructions description (latency, uops dispatch...)
  - Machine model

- For a given binary and micro-architecture, provides
  - Quality metrics (how well the binary fits a uarch)
  - Static performance (lower bounds on cycles)
  - Hints and workarounds to improve static performance
Static analysis

Sandy Bridge Pipeline Model
Static analysis

Key metrics

- Unrolling (unroll factor detection)
  - Allows to statically predict performance for different unroll factors

- Vectorization (ratio and speedup)
  - Allows to predict vectorization (if possible) speedup and increase vectorization ratio if it’s worth

- High latency instructions (division and square root)
  - Allows to use less precise but faster instructions like RCP (1/x) and RSQRT (1/sqrt(x))
Section 1.1.1: Source loop ending at line 7

---

Composition and unrolling

It is composed of the loop 0 and is **not unrolled or unrolled with no peel/tail code** (including vectorization).

Type of elements and instruction set

- 3 SSE or AVX instructions are processing **single precision FP elements in scalar mode** (one at a time).

Vectorization

Your loop is **not vectorized** (all SSE/AVX instructions are used in scalar mode).

Matching between your loop... and the binary loop

The binary loop is composed of **1 FP arithmetical operations:**

1. divide

The binary loop is loading 8 bytes (2 single precision FP elements).

The binary loop is storing 4 bytes (1 single precision FP elements).

Arithmetic intensity is **0.08 FP operations per loaded or stored byte.**

Cycles and resources usage

Assuming all data fit into the L1 cache, each iteration of the binary loop takes **14.00 cycles**.

At this rate:

- **0% of peak computational performance** is reached (0.07 out of 16.00 FLOP per cycle (GFLOPS @ 1GHz))
- **1% of peak load performance** is reached (0.57 out of 32.00 bytes loaded per cycle (GB/s @ 1GHz))
- **1% of peak store performance** is reached (0.29 out of 16.00 bytes stored per cycle (GB/s @ 1GHz))
Pathological cases

Your loop is processing FP elements but is NOT OR PARTIALLY VECTORIZED.

Since your execution units are vector units, only a fully vectorized loop can use their full power.

By fully vectorizing your loop, you can lower the cost of an iteration from 14.00 to 3.50 cycles (4.00x speedup).

Two propositions:

- Try another compiler or update/tune your current one:
  * gcc: use O3 or Ofast. If targeting IA32, add mfpmath=sse combined with march=\(<\text{cputype}\), msse or msse2.
  * icc: use the vec-report option to understand why your loop was not vectorized. If "existence of vector dependences", try the IVDEP directive. If, using IVDEP, "vectorization possible but seems inefficient", try the VECTOR ALWAYS directive.
- Remove inter-iterations dependences from your loop and make it unit-stride.

WARNING: Fix as many pathological cases as you can before reading the following sections.

Bottlenecks

The divide/square root unit is a bottleneck.

Try to reduce the number of division or square root instructions.

If you accept to lose numerical precision, you can speedup your code by passing the following options to your compiler:

- gcc: (ffast-math or Ofast) and mrecip
- icc: this should be automatically done by default

By removing all these bottlenecks, you can lower the cost of an iteration from 14.00 to 1.50 cycles (9.33x speedup).
Outline

- Introduction
- MAQAO Tool
- Static Analysis
- Dynamic Analysis
- Building performance evaluation tools
- Conclusion
Dynamic analysis

- Static analysis is optimistic
  - Data in L1$
  - Believe architecture

- Get a real image:
  - Coarse grain: find hotspots
  - Fine grain: specialized modules
    - Memory
    - Code quality
  - MIL: specialized instrumentation
Dynamic analysis
Detecting hotspots

- Most time consuming functions and loops
  - Sampling method with HPCs: low overhead
  - Handles serial and OpenMP applications (next target = MPI).
  - Exclusive % time
Dynamic analysis
Detecting hotspots

- Functions

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Time %</th>
</tr>
</thead>
<tbody>
<tr>
<td>compute_rhs_omp#region#1</td>
<td>25.35</td>
</tr>
<tr>
<td>binvcrhs_</td>
<td>22.73</td>
</tr>
<tr>
<td>matmul_sub_</td>
<td>11.27</td>
</tr>
<tr>
<td>y_solve_omp#loop#1</td>
<td>10.65</td>
</tr>
<tr>
<td>z_solve_omp#loop#1</td>
<td>9.82</td>
</tr>
<tr>
<td>x_solve_omp#loop#1</td>
<td>8.58</td>
</tr>
<tr>
<td>__kmp_wait_sleep [libiomp5.so]</td>
<td>3.89</td>
</tr>
<tr>
<td>matvec_sub_</td>
<td>2.82</td>
</tr>
<tr>
<td>add_omp#loop#1</td>
<td>2.19</td>
</tr>
<tr>
<td>__kmp_x86_pause [libiomp5.so]</td>
<td>1.26</td>
</tr>
<tr>
<td>Others</td>
<td>0.71</td>
</tr>
<tr>
<td>lhsinit_</td>
<td>0.21</td>
</tr>
<tr>
<td>__kmp_yield [libiomp5.so]</td>
<td>0.18</td>
</tr>
<tr>
<td>binvrhs_</td>
<td>0.12</td>
</tr>
<tr>
<td>exact_solution_</td>
<td>0.12</td>
</tr>
<tr>
<td>exact_rhs_omp#region#1</td>
<td>0.05</td>
</tr>
<tr>
<td>initialize_omp#region#1</td>
<td>0.02</td>
</tr>
<tr>
<td>__kmpc_barrier [libiomp5.so]</td>
<td>0.01</td>
</tr>
<tr>
<td>__kmpc_for_static_init_4 [libiomp5.so]</td>
<td>0.01</td>
</tr>
<tr>
<td>__kmp_launch_thread [libiomp5.so]</td>
<td>0.01</td>
</tr>
<tr>
<td>__kmp_run_before_invoked_task [libiomp5.so]</td>
<td>0.00</td>
</tr>
<tr>
<td>data.10538 [libc-2.12.so]</td>
<td>0.00</td>
</tr>
<tr>
<td>kmp_threadprivate_insert [libiomp5.so]</td>
<td>0.00</td>
</tr>
<tr>
<td>__kmpc_for_static_fini [libiomp5.so]</td>
<td>0.00</td>
</tr>
<tr>
<td>__kmp_task_team_sync [libiomp5.so]</td>
<td>0.00</td>
</tr>
<tr>
<td>__kmp_invoke_microtask [libiomp5.so]</td>
<td>0.00</td>
</tr>
</tbody>
</table>
### Dynamic analysis

#### Detecting hotspots

- **Loops**

<table>
<thead>
<tr>
<th>#</th>
<th>Loop ID</th>
<th>Source Infos</th>
<th>Level</th>
<th>Time %</th>
</tr>
</thead>
<tbody>
<tr>
<td>#</td>
<td>111</td>
<td>compute_rhs_omp#region1 - 291,336</td>
<td>Innermost</td>
<td>6.12</td>
</tr>
<tr>
<td>#</td>
<td>181</td>
<td>y_solve_omp#loop#1 - 136,298</td>
<td>Innermost</td>
<td>4.33</td>
</tr>
<tr>
<td>#</td>
<td>191</td>
<td>z_solve_omp#loop#1 - 137,299</td>
<td>Innermost</td>
<td>4.35</td>
</tr>
<tr>
<td>#</td>
<td>171</td>
<td>x_solve_omp#loop#1 - 137,299</td>
<td>Innermost</td>
<td>4.07</td>
</tr>
<tr>
<td>#</td>
<td>182</td>
<td>y_solve_omp#loop#1 - 46,128</td>
<td>Innermost</td>
<td>3.77</td>
</tr>
<tr>
<td>#</td>
<td>192</td>
<td>z_solve_omp#loop#1 - 46,128</td>
<td>Innermost</td>
<td>3.23</td>
</tr>
<tr>
<td>#</td>
<td>96</td>
<td>compute_rhs_omp#region1 - 375,379</td>
<td>Innermost</td>
<td>3.24</td>
</tr>
<tr>
<td>#</td>
<td>149</td>
<td>compute_rhs_omp#region1 - 70,119</td>
<td>Innermost</td>
<td>3.08</td>
</tr>
<tr>
<td>#</td>
<td>133</td>
<td>compute_rhs_omp#region1 - 181,225</td>
<td>Innermost</td>
<td>2.95</td>
</tr>
<tr>
<td>#</td>
<td>161</td>
<td>compute_rhs_omp#region1 - 26,36</td>
<td>Innermost</td>
<td>2.90</td>
</tr>
<tr>
<td>#</td>
<td>156</td>
<td>compute_rhs_omp#region1 - 52,53</td>
<td>Innermost</td>
<td>2.48</td>
</tr>
<tr>
<td>#</td>
<td>172</td>
<td>x_solve_omp#loop#1 - 48,130</td>
<td>Innermost</td>
<td>2.51</td>
</tr>
<tr>
<td>#</td>
<td>198</td>
<td>add_omp#loop#1 - 22,23</td>
<td>Innermost</td>
<td>2.17</td>
</tr>
<tr>
<td>#</td>
<td>176</td>
<td>y_solve_omp#loop#1 - 386,389</td>
<td>InBetween</td>
<td>1.14</td>
</tr>
<tr>
<td>#</td>
<td>141</td>
<td>compute_rhs_omp#region1 - 144,148</td>
<td>Innermost</td>
<td>1.19</td>
</tr>
<tr>
<td>#</td>
<td>186</td>
<td>z_solve_omp#loop#1 - 399,402</td>
<td>InBetween</td>
<td>1.20</td>
</tr>
<tr>
<td>#</td>
<td>166</td>
<td>x_solve_omp#loop#1 - 387,390</td>
<td>InBetween</td>
<td>0.99</td>
</tr>
<tr>
<td>#</td>
<td>79</td>
<td>compute_rhs_omp#region1 - 419,420</td>
<td>InBetween</td>
<td>0.83</td>
</tr>
<tr>
<td>#</td>
<td>121</td>
<td>compute_rhs_omp#region1 - 252,256</td>
<td>Innermost</td>
<td>0.78</td>
</tr>
<tr>
<td>#</td>
<td>189</td>
<td>z_solve_omp#loop#1 - 399,402</td>
<td>Innermost</td>
<td>0.44</td>
</tr>
<tr>
<td>#</td>
<td>179</td>
<td>y_solve_omp#loop#1 - 386,389</td>
<td>Innermost</td>
<td>0.34</td>
</tr>
<tr>
<td>#</td>
<td>165</td>
<td>x_solve_omp#loop#1 - 387,390</td>
<td>Innermost</td>
<td>0.38</td>
</tr>
<tr>
<td>#</td>
<td>112</td>
<td>compute_rhs_omp#region1 - 291,336</td>
<td>Innermost</td>
<td>0.41</td>
</tr>
<tr>
<td>#</td>
<td>185</td>
<td>z_solve_omp#loop#1 - 399,402</td>
<td>Innermost</td>
<td>0.28</td>
</tr>
<tr>
<td>#</td>
<td>169</td>
<td>x_solve_omp#loop#1 - 387,390</td>
<td>Innermost</td>
<td>0.38</td>
</tr>
<tr>
<td>#</td>
<td>175</td>
<td>y_solve_omp#loop#1 - 386,389</td>
<td>Innermost</td>
<td>0.30</td>
</tr>
<tr>
<td>#</td>
<td>188</td>
<td>z_solve_omp#loop#1 - 399,402</td>
<td>Innermost</td>
<td>0.19</td>
</tr>
<tr>
<td>#</td>
<td>168</td>
<td>x_solve_omp#loop#1 - 387,390</td>
<td>Innermost</td>
<td>0.21</td>
</tr>
<tr>
<td>#</td>
<td>134</td>
<td>compute_rhs_omp#region1 - 181,225</td>
<td>Innermost</td>
<td>0.20</td>
</tr>
<tr>
<td>#</td>
<td>150</td>
<td>compute_rhs_omp#region1 - 70,119</td>
<td>Innermost</td>
<td>0.15</td>
</tr>
<tr>
<td>#</td>
<td>170</td>
<td>x_solve_omp#loop#1 - 333,353</td>
<td>Innermost</td>
<td>0.17</td>
</tr>
</tbody>
</table>

...
Dynamic analysis
Memory behavior characterisation

- Transformation opportunities, e.g.: loop interchange
- Data reshaping opportunities, e.g.: array splitting
Memory behavior characterization

Single threaded aspects: Inefficient patterns

Real code example: PMBENCH

```plaintext
for (int n=0; n<M; n++)
    if (lambdaz[n] > 0.) {
        for (int j=0; j<mesh.NCx; j++)
            for (int i=1; i<mesh.NCz; i++)
                J_upz[IDX3C(n,i,M,j,(mesh.NCz+1)*M)] = Jz[IDX3C(n-1,M,j,(mesh.NCz)*M)] * lambdaz[n];
    }
    if (lambdaz[n] < 0.){
```

MTL output

- **Load (Double) - Pattern:** 8*i1  (Hits : 100% | Count : 1)
- **Load (Double) - Pattern:** 8*i1+217600*i2+1088*i3  (Hits : 100% | Count : 1)
- **Store (Double) - Pattern:** 8*i1+218688*i2+1088*i3  (Hits : 100% | Count : 1)

- Stride 1 (8/8) one access for outmost
- Poor access patterns for two instructions
- Idealy: smallest strides inside to outside
- Here: interchange n and i loops
Memory behavior characterization

Single threaded aspects: Inefficient patterns

Real code example: PMBENCH

- Example: `flux_numerique_z`, loop 193 (same for 195)
- Same kind of optimization for loops 204 and 206

```
for (int n=0; n<M; n++) {
    if (lambdaz[n] > 0.){
        for (int j=0; j<NCx; j++)
            for (int i=1; i<NCz; i++) // loop 193
                J_upz[IDX3C(n,i,M,j,(NCz+1)*M)]=
                    Jz[IDX3C(n,i-1,M,j,(NCz)*M)] * lambdaz[n];
    }
    if (lambdaz[n] < 0.)
        ...//loop 195
}
```

```
for (int j=0; j<NCx; j++)
    for (int n=0; n<M; n++) {
        if (lambdax[n] > 0.){
            for (int i=1; i<NCz; i++) // loop 193
                J_upz[IDX3C(n,i,M,j,(NCz+1)*M)]=
                    Jz[IDX3C(n,i-1,M,j,(NCz)*M)] * lambdaz[n];
        }
        if (lambdaz[n] < 0.)
            ...//loop 195
    }
```

7.7x local speedup (loops) ➔ 1.4x GLOBAL speedup
Introduction

MAQAO Tool

Static Analysis

Dynamic Analysis

Building performance evaluation tools

Conclusion
MIL: Instrumentation language
New DSL

- A domain specific language to easily build tools
- Fast prototyping of evaluation tools
  - Easy to use ➔ easy to express ➔ productivity
  - Focus on what (research) and not how (technical)
- Coupling static and dynamic analyses
- Static binary instrumentation (rewriting)
  - Efficient: lowest overhead
  - Robust: ensure the program semantics
  - Accurate: correctly identify program structure
- Drive binary manipulation layer
Instrumentation language

Language concepts/features

- Global variables (in binary)
- Events
- Filters
- Actions
- Configuration features
  - Output
  - Language behavior (properties)
- Runtime embedded code
Instrumentation Language
Language concepts/features

Events: Where?

<table>
<thead>
<tr>
<th>Level</th>
<th>Events</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program</td>
<td>Entry / Exit (avoid LD + exit handlers)</td>
</tr>
<tr>
<td>Function</td>
<td>Entries / Exits</td>
</tr>
<tr>
<td>Loop</td>
<td>Entries / Exits / Backedges</td>
</tr>
<tr>
<td>Block</td>
<td>Entry / Exit</td>
</tr>
<tr>
<td>Instruction</td>
<td>Before / After</td>
</tr>
<tr>
<td>Callsite</td>
<td>Before / After</td>
</tr>
</tbody>
</table>
Probes: What?

- External functions
  - Name
  - Library
  - Parameters: int, string, macros, function (static ↔ dynamic)
  - Return value
  - Demangling
  - Context saving

- ASM inline: gcc-like

- Runtime embedded code (lua code within MIL file)
Filters:

- Why? Reduce instrumentation probes
  - Target what really matters

- Lists: regular expressions
  - White list
  - Black list

- Built-in: structural properties attributes
  - Example: nesting level for a loop

- User defined: an action that returns true/false
Instrumentation Language

Language concepts/features

- **Actions:**
  - Why? For complex instrumentation queries
  - Scripting ability (Lua code)
  - User-defined functions
  - Access to MAQAO Plugins API (existing modules)
Passes:

- To address complex multistep instrumentations

Example: detect OpenMP events

  - Step 1: static analysis to detect sequences of call sites
    - Only events and actions are used
  
  - Step 2: instrument
    - Select (same or new) events and insert probes based on step 1
Conclusion

- Select a consistent methodology

- Detect hotspots

- Use specialized modules at loop level
  - Assess code quality through static analysis
  - Memory behavior characterization

- If no relevant existing module: use MIL

- Iterative approach
Thanks for your attention!

Questions?
Hand-on exercises

- Refer to the exercises folder
  - cd $UNITE_ROOT/tutorial/maqao/
  - Read Outline.txt