Application Characterization: A comprehensive way of analyzing and understanding interaction between hardware and software (applications/compiler/runtime): performance and energy

Jean Christophe Beyler, Intel ECR

BOF Submitted by:
Marie-Christine Sawley, Intel ECR & Gregori Fursin, INRIA
ECR Team

ECR team includes over 25 high level researchers

ECR Team is part of research labs in Europe and North America

ECR is a member of the Intel EMEA HPC Exascale labs together with ExaScience lab and ExaCluster lab

Part of DSG-DCSG
2 main Streams of Research

Software for application characterization and performance optimization

*Extract fine grain information about the interaction of whole software with the underlying architecture*

Application co-design

*Leveraging from the low level information and the capacity of new architectures for enabling progress in science using computational power*
Outline

1. Full Application

2. Small Representative Codelets

3. Coarse Grain Tools
   MAQAO, DECAN
   Codelet Profiles
   Optimization Opportunities

4. Underlying Architecture
   MicroBenchmarks
   MDL

5. Handling all the Data
   ASK
   Machine Learning

6. Capacity and Prediction Models

7. Tying it all together: CTI
Cutting the Application Up

• First step: finding the hot spots

• Considering full applications is difficult
  • Study the hotspots
  • Automatic solution:
    • Using CAPS Enterprise’s Codelet Finder tool

• Second step: work on the codelets separately
Codelet Finder

- Key features
  - Implemented by CAPS Enterprise
  - Handles C or Fortran codes
  - Automatically detects hotspots and extracts loops into:
    - Kernel, wrapper, data input
    - Data input is retrieved by a core dump before the kernel
- Future work
  - Allow users to modify input data easily
  - Add more supported constructs
2. Small Representative Codelets

3. Coarse Grain Tools
   MAQAO, DECAN
   Codelet Profiles
   Optimization Opportunities
Tools to Quantify

• Logical next step
• Measure performance with profiling tools such as MAQAO
  • Provides important information
  • What can be optimized?
  • What is obtainable?

Interactive Control Flow Graph (Exagraph)

Loop Structure and Assembly Code
• Disassemble or reassemble SSE and AVX binaries
• Performance model for Core2, Nehalem, and Sandy Bridge
• Low overhead profiler
  • < 100 cycles per probe
  • OMP compliant
• Performance analysis
  • No pragma or source code alteration
  • Vectorization ratio
  • Detailed pipeline model:
    • Dispatch, decoder, LSD, per port pressure
  • Memory traffic
  • Aggregate memory instructions per group
  • Unrolling factor
• Static performance prediction
  • 'What if' the code is fully vectorized
  • 'What if' the data is stored in L1
• Modular Architecture
  • Demo of TAU using binary capacities of MAQAO at SC11
QMC == CHEM with MAQAO

\[
\begin{align*}
\text{DIR$ VECTOR ALIGNED} \\
\text{do } j=1, \text{LDC} \\
C1(j,i) &= C1(j,i) + (A(j, k_{\text{vec}}(1)) \times d11 & + A(j, k_{\text{vec}}(2)) \times d21 & + A(j, k_{\text{vec}}(3)) \times d31 & + A(j, k_{\text{vec}}(4)) \times d41) \\
C2(j,i) &= C2(j,i) + (A(j, k_{\text{vec}}(1)) \times d12 & + A(j, k_{\text{vec}}(2)) \times d22 & + A(j, k_{\text{vec}}(3)) \times d32 & + A(j, k_{\text{vec}}(4)) \times d42) \\
\text{enddo}
\end{align*}
\]

Dealing with the two hottest loops in the application
- Dense x sparse matrix multiply

FLOP/cycle not optimal:
- 12.8 but should be 16
  - AVX, 32 bits elements, perfect ADD/MUL balance

Replacing LDC with its value "hard coded" allows the compiler to factor for the two matrices C1 and C2

MAQAO static analysis before (top) and after (bottom) optimization
Tools to Explain

• MAQAO and similar tools provide information
  • Profiler detects hot spot
  • MAQAO goes beyond and evaluates the gap
    • Current and optimal static performance
    • It remains the discrepancy is difficult to understand

• DECAN is an exploratory tool
DECAN

• DECAN’s concept is simple
  • Measure the original binary
  • Patch and replace the selected instructions group in the original binary
  • New binary is generated for each patch
  • Measure new binaries
  • Measurements are represented in a CSV file
    • Analyze and compare
DECAN

• Codelet Decomposition
  • MISTREAM
    • All vector arithmetic instructions are deleted
  • FPSTREAM
    • All vector loads and store instructions are deleted
  • NOFPNOMISTREAM
    • All vector arithmetic, load, and store instructions are deleted

Codelet Contains: Memory Instruction Arithmetic Instruction Branch Instruction

Version without Memory Instruction

Version without Arithmetic Instruction

Version with Only Branch Instruction

Results Can Be Analyzed Separately
MAQAO + DECAN Provides

• Speed-up by a factor of 4
  • Up to 37% of the peak performance on Sandy Bridge

• Vectorization ratio crucial on Sandy Bridge

• Value profiling
Outline

2. Small Representative Codelets

3. Coarse Grain Tools
   MAQAO, DECAN
   Codelet Profiles
   Optimization Opportunities

4. Underlying Architecture
   MicroBenchmarks
   MDL
Underlying Architecture

- Understanding the target architecture
  - Gives insight on potential bottlenecks
  - Provides solutions to optimize a code

- How is it done?
  - Emulators or simulators are slow if even available
  - Microbenchmarking considers the hardware as a black box
MicroTools

- Template XML Format
- MicroCreator
  - Source-to-Source Compiler
- MicroLauncher
  - Generic
  - Multi-Process Benchmark Launcher

- Generated Assembly or C Code
- Performance Information
- Energy Information
MicroTools Usage

MicroTools enables an exhaustive exploration of architecture performance.

1) Two real case-studies
   - NOPS impact on dispatcher
   - Memory + arithmetic interaction

2) How to deal with all the data collected?
   - Automated reports analysis
NOPS Experiment

• Goal: Evaluate the dispatch unit

• Loop body parameters
  • The NOP instruction size:
    • Varies from 1 to 9 bytes
  • The number of NOP instructions in the loop body:
    • Varies from 4 to 32

• Each loop body tested consists of the same NOP instructions repeated from 4 to 32 times
NOPS Experiment

![Graph showing dispatch performance for NOPS Nehalem]
NOPS Experiment

![NOPS Experiment Graph](image)
NOPS Experiment

- Dispatcher capacity is around 3 nops/cycle
- For 1-byte and 2-byte nops
  - Dispatch behavior is linear
- For large size nops
  - Dispatch rate falls down to 0.96 instructions per cycle
Outline

2. Small Representative Codelets

3. Coarse Grain Tools
    MAQAO, DECAN
    Codelet Profiles
    Optimization Opportunities

4. Underlying Architecture
    Micro Benchmarks
    MDL

5. Handling all the Data
    ASK
    Machine Learning

Machine Learning
HANDLING THE DATA

- MicroCreator philosophy:
  - Exhaustively search around a given program specification
  - Sometimes, we need to reduce the space (ASK)
- A lot of data produced
  - We need automatic data handling tools
    - Validate the stability of the results
    - Identify unexpected situations to help the engineer.
ASK (Adaptive Sampling Kit)

- ASK is a toolkit providing state of the art sampling strategies
- Modular and Extensible Pipeline:
  - Combine different static and dynamic strategies
  - Easy to add new custom sampling strategy
  - Easy integration with benchmarking tools

ASK contains multiple sampling point selections
ASK selects points to test until the studied variance is “acceptable” by the user selections
ASK provides the approximate domain space analysis

User defines the domain space with an easy plugin
ASK: An example

- Search the domain space
- Find high variance regions
- Draw new points from high-variance regions

“Flat” regions are less interesting to explore.
Order Influence Report

- Decomposes results per number of stores and loads
- Quickly identify configurations where performance depends on the order of instructions

<table>
<thead>
<tr>
<th>MOVAPS Sandybridge L3</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>#L \ #S</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>1</td>
<td>7.29</td>
<td>7.6</td>
<td>17.85</td>
<td>19.15</td>
<td>24.29</td>
<td>33.07</td>
<td>38.22</td>
</tr>
<tr>
<td>2</td>
<td>9.8</td>
<td>10.6</td>
<td>20.19</td>
<td>20.86</td>
<td>26.16</td>
<td>34.89</td>
<td>31.21</td>
</tr>
<tr>
<td>3</td>
<td>12.88</td>
<td>13.18</td>
<td>21.46</td>
<td>22.81</td>
<td>27.92</td>
<td>37.55</td>
<td>33.22</td>
</tr>
<tr>
<td>4</td>
<td>15.67</td>
<td>15.74</td>
<td>23.08</td>
<td>24.35</td>
<td>28.43</td>
<td>39.41</td>
<td>33.81</td>
</tr>
<tr>
<td>5</td>
<td>18.56</td>
<td>19.01</td>
<td>24.89</td>
<td>25.7</td>
<td>30.55</td>
<td>41.53</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>21.42</td>
<td>21.54</td>
<td>26.88</td>
<td>28.25</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>24.55</td>
<td>25.67</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Outline

2. Small Representative Codelets

5. Handling all the Data
   - ASK
   - Machine Learning

6. Capacity and Prediction Models
Capacity Model

Application → Codelet Extraction Tool (Codelet Finder) → Codelets → Specific Memory Assembly Instructions Set

Codelets → Macro Stream Generation Tool (DECAN) → Macros
- Floating Point Only
- Memory Only
- Memory With 1BYTE NOPS
- No Floating Point No Memory

Macros → Benchmarking

Benchmarking → Single-rate Virtual Nodes Analysis (SV-Nodes Analysis) → Synthetic Data

Hardware Performance Prediction
Capacity Model

• Capacity Model and MDL
  • Provide prediction and modelization of program performance

• However, alone the tools are less valuable
  • A need to centralize the data and analysis:

CTI : Codelet Tuning Infrastructure
Outline

1. Full Application

2. Small Representative Codelets
   - Small Codelets
   - MicroBenchmarks
   - MDL

3. Coarse Grain Tools
   - MAQAO, DECAN
   - Codelet Profiles
   - Optimization Opportunities

4. Underlying Architecture

5. Handling all the Data
   - ASK
   - Machine Learning

6. Capacity and Prediction Models

7. Tying it all together: CTI
Codelet Tuning Infrastructure (CTI)

- A single place to store a huge amount of data
  - File manager
    - File sharing, updating, processing, viewing
  - Codelet manager
  - CSV automatic file insertion
  - Query the data
- Automate experiments
- Tools integrator

Legend

- [ ] To Be Done
- [x] Done

CTI

- Files
- Codelet Finder
- MicroTools
- DECAN
- MAQAO
- Experiment Data
CTI Codelet Manager

- Codelet Manager
  - Daemon
    - Asynchronous
    - Registering Codelets
  - C.T.I.
    - User
  - Codelet Finder

- Application
  - Codelets

- Front end
  - M1
  - M2
  - Etc…
CTI Codelet Manager

• Launching the « create » process

```
johndoe@toto:$ cti application create_codelets test borodine
The process is running!
DATA_CTI_UID=53e7cbd5-f429-4d40-aff6-c767ca3abc7a
```
CTI Codelet Manager

• Checking the codelets list

```
c01a8629-c459-4fed-8ce0-b0cd41b1b44b

Repository = /home/users/ftalbart/test/.ctr/data/c01a8629-c459-4fed-8ce0-b0cd41b1b44b

UID = c01a8629-c459-4fed-8ce0-b0cd41b1b44b

Plugin command = init
```

```
cti view data <UID|Alias>
```

johndoe@toto:~$ cti view data <UID|Alias>
CTI Codelet Manager

• Checking the codelet files

```
john@toto:~ $ cti import_files get <UID> Aliases
```
**CTI MAQAO Integration**

1) **Codelet to Binary**
   - The codelet is sent to the cluster for compilation.

2) **Binary to CSV**
   - Binary is analyzed using MAQAO and a CSV file with results is produced.

3) **CSV to CTI entry**
   - CSV file is loaded into CTI repository and saved in an entry.
CTI MAQAO Integration

CSV File

Loop 1 Analysis
- Number of instructions
- Number of micro operations
- Size (in bytes) of the loop
- Number of XMM or YMM used registers
- Does the loop fit in cache?
- Number of bytes loaded and stored
- Number of micro-operations per iteration for each execution port
- Number of cycle per iteration for each execution port
- Vectorization ratios for:
  - Loads
  - Store
  - Add/Sub
  - Mul
  - Others
- Predictions if data fit in L1 cache:
  - Number of cycles
  - FLOP/cycle
  - Bytes loaded/cycle
  - Bytes stored/cycle
  - Number of cycle if fully vectorized

Loop 2 Analysis

Loop n Analysis
Overall Conclusions

• Studying a large application is difficult
  • Dividing the application into codelets
  • Using tools such as MAQAO and DECAN help understand the codelet’s performance and behavior
  • Understanding the underlying architecture with MicroTools and the MDL help detect hardware bottlenecks
  • Analyzing all the data is only possible with automatic tools and infrastructures such as CTI
Acknowledgements

• Pablo Oliviera, UVSQ
• José Noudohouenou, UVSQ
• Franck Talbart, UVSQ
• William Jalby, CTO, UVSQ
• Jean-Thomas Acquaviva, Lead of the Performance Tools
• Bettina Krammer, Head of Software Tools, UVSQ
• Marie-Christine Sawley, Lab Director, Intel
ECR Contacts

• Address
UVSQ, 45 Av. des Etats-Unis, Buffon building, 5th floor
78 000 Versailles, France

• Web site: www.exascale-computing.eu

• Team
William Jalby, CT , william.jalby@uvsq.fr
Jean Christophe Beyler, Lead of Application Characterization,
jean.christophe.beyler@intel.com
Marie-Christine Sawley, Co-design, marie-christine.sawley@intel.com
Bettina Krammer, Tools, bettina.krammer@uvsq.fr

• Collaboration partners

---

Exascale computing research
Thank you
DECAN + MICROTOOLS

saxpy2MIS

1-core execution, Low Frequency

Cycles per iteration vs Number of Nops
DECAN + MICROTOOLS

saxpy2MIS

12-core execution, Low Frequency

Cycles per iteration vs. Number of Nops
DECAN + MICROTOOLS

saxpy2MIS

12-core execution, High Frequency

Cycles per iteration vs. Number of Nops

- L1
- L2
- L3
- RAM
Handling Stores

• L3 Nehalem
  • Pure load patterns scale better with frequency
Handling Stores

- L3 Nehalem
  - Pure load patterns scale better with frequency

MOVSD

2 LOADS  2 LOADS + 1 STORE
Handling Stores

- L3 Nehalem
  - Pure load patterns scale better with frequency
Handling Stores

• L3 Sandy Bridge cache scales much better
• Perfect scaling for all the store and load configurations
Handling Stores

- L3 Sandy Bridge cache scales much better
- Perfect scaling for all the store and load configurations

MOVSD

<table>
<thead>
<tr>
<th>2 LOADS</th>
<th>2 LOADS + 1 STORE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1/3.3 1/2.2 1/1.6</td>
<td>1/3.3 1/2.2 1/1.6</td>
</tr>
</tbody>
</table>
Handling Stores

- L3 Sandy Bridge cache scales much better
- Perfect scaling for all the store and load configurations
Energy
Results

Used a dual-Nehalem, 6-cores each, nine possible frequencies
Used a Sandy Bridge Quad-core, sixteen frequencies
Compared with O3 execution, the Linux governors and static frequencies
Gromacs
Nehalem (Energy)

![Graph showing energy and time spent against frequencies (KHz).]
RTM on Nehalem

RTM Power and Performance

Only 2% difference!
Ratio Turboboost on REST

- Energy
- Performance

Turboboost (J)
---------------------
Rest (J)

Turboboost (s)
---------------
Rest (s)

Rest = Turboboost
What about Sandy bridge?

Libquantum
Sandy Bridge (Energy)

- Time Spent (hh:mm:ss)
  - Frequencies (KHz)
  - Energy (Joules)

Exascale computing research
What about Watts?

**Watt Usage on Sandy Bridge SPEC2006**

![Graph showing watt usage vs frequencies for various applications like libquantum, gromacs, bzip2, calculix, lbm, perlbench, milc, gobmk, and h264ref.]
Sandy Bridge’s Hardware counters 1/2

QMC=Chem
Sandy Bridge (Power)

Exascale computing research

64
Sandy Bridge’s Hardware counters 2/2

QMC=Chem
Sandy Bridge (Energy)

Frequencies (KHz)

Time Spent (hh:mm:ss)

Energy Core (J)

Energy Total (J)

Exascale computing research

65
UpDownbench profiled with Eprof
(Hardware counter from PAPI)