FOR AARCH64 ARCHITECTURES

Hugo BOLLORE - hugo.bollore@uvsq.fr

ARM RESEARCH SUMMIT 2018 – September 17-19 – Cambridge, UK
SUMMARY

- The MAQAO framework, a performance analysis and optimisation tool
- Static capabilities and the Code Quality Analyzer module
- Port for ARM 64-bit applications
- Using MAQAO to study ARM HPC compiler vectorisation
MAQAO: Modular Assembly Quality Analyzer and Optimizer

- Objectives:
  - Characterizing performance of applications
  - Focusing on performance at the **core level**
  - **Guiding users** through optimisation process

- Characteristics:
  - Modular tool offering complementary views
  - Support for **x86-64, Xeon Phi, AArch64** architectures
  - LGPL3 Open Source software
MAQAO: Analysis at binary level

- Advantages:
  - Takes into account compilers optimisations/transformations and ISA specificities
  - Allows evaluation of the executed binary code
  - No recompilation necessary

- Requirements:
  - Debug information to correlate analyses with source code
MAQAO: STRUCTURE

- LProf (Lightweight Profiler): A sampling based profiler for identifying application hotspots.
- VProf (Value Profiler): A dynamic value profiler. Extracts parameter values from subroutine calls.
- DECAN (DECremental Analyzer): A binary rewriting system to evaluate performance impact of specific sequences of instructions.
- OneView (Performance Views Aggregator): A module automating the analysis process by driving the other modules, aggregating the results in a hierarchical way, and providing useful hints to restructure code.
CQA: Code Quality Analyzer

- **Goal:** Assist developers in improving code performance
- **Features:**
  - Evaluates the *quality* of the compiler generated code
  - Returns *hints and workarounds* to improve quality
  - Targets *compute-bound* codes
- **Static analysis:**
  - Requires *no execution* of the application
  - Allows *cross-platform analyses*
CQA: Motivating example

1. High number of statements
2. Non-unit stride access
3. Indirect accesses
4. DIV/SQRT
5. Reductions
6. Variable number of iterations

Source code and associated issues

```
do j = ni + nvalue1, nato 6) Variable number of iterations
  nj1 = ndim3d*j + nc ; nj2 = nj1 + nvalue1 ; nj3 = nj2 + nvalue1
  u1 = x11 - x(nj1) ; u2 = x12 - x(nj2) ; u3 = x13 - x(nj3)
  rtest2 = u1*u1 + u2*u2 + u3*u3 ; cnij = eci*Qold(j)
  rij = demi*(rvwi + rvwal1(j))
  drtest2 = cnij*(rtest2 + rij) ; drtest = sqrt(drtest2)
  Eq = qq1*qq(j)*drtest
  ntj = nti + ntype(j)
  Ed = ceps(ntj)*drtest2*drtest2*drtest2
  Eqc = Eqc + Eq ; Ephob = Ephob + Ed
  gE = (c6*Ed + Eq)*drtest2 ; virt = virt * gE * rtest2
  u1g = u1*gE ; u2g = u2*gE ; u3g = u3*gE
  g1c = g1c - u1g ; g2c = g2c - u2g ; g3c = g3c - u3g
  gr(nj1, thread_num) = gr(nj1, thread_num) + u1g
  gr(nj2, thread_num) = gr(nj2, thread_num) + u2g
  gr(nj3, thread_num) = gr(nj3, thread_num) + u3g
```

2) Non-unit stride accesses
3) Indirect accesses
4) DIV/SQRT
5) Reductions
6) Variable number of iterations
MAQAO FOR AARCH64 ARCHITECTURES - ARM RESEARCH SUMMIT (17-19 SEP 2018)
CQA: Application on Motivating Example

Source code and associated issues

1. High number of statements
2. Non-unit stride access
3. Indirect accesses
4. DIV/SQRT
5. Reductions
6. Variable number of iterations
ARM architectures

- ARM ecosystem is growing fast:
  - Several manufacturers building processors tuned for HPC
  - Mathematical libraries ported for ARMv8 processors
  - Several supercomputers under construction

- A need for optimisation
  - The Mont-Blanc 3 project:
    - Development of the software ecosystem for HPC
    - Port of optimisation tools: MAQAO for ARMv8 available at https://github.com/maqao/maqao
CQA: Architecture model validation

- Comparison of the static simulation against measurements:
    - Representative of many scientific kernels
    - Target different architectural and performance bottlenecks
  - Environment:
    - ARM Cortex A-57
    - Compiled with gcc 6.3.0
    - Data in L1 cache
CQA: Accuracy on Numerical Recipes

- Stride 1
- Average accuracy: 89%
CQA: Accuracy on Numerical Recipes

- Non unit-stride
- Average accuracy: 92%
VECTORISATION: ARMv8 SVE

- **Scalable Vector Extension:**
  - Compatibility across implementations (embedded systems, big data, HPC, …)
  - Advantages for High Performance Computing:
    - Forward compatibility
    - Less micro-architecture specific processing for the compilers
    - Predicates allowing to work on partial vectors
- Compilers have to produce vectorised code in order to benefit from SVE
VECTORISATION: Study of the Arm HPC compiler

- Test Suite for Vectorizing Compiler (Maleki et al. 2011 - *An evaluation of vectorising compilers.)*
  - Covers several aspects and check handling of techniques serving auto-vectorisation
  - Growing complexity
- Intel:
  - Intel compiler *icc 17.0.2*
  - Compiled for a Intel Core i7-4770 (Haswell)
- Arm:
  - Arm HPC tools *armclang 18.2 (based on llvm 5.0.1)*
  - Compiled for a Cavium ThunderX2 (ARMv8)
- Using CQA to compute the percentage of vectorised code generated by the compilers:
  \[
  \frac{\sum \text{Number of vectorised instructions}}{\sum \text{Number of vectorisable instructions}}
  \]
VECTORISATION: Linear dependence

- Compilers must check that there is no dependence between iterations

- for (int i = 1; i < LEN; i += 2)
  \[ a[i] = a[i - 1] + b[i]; \]

- False dependency:
  - Accessing \( a[i - 1] \) does not generate a dependency with \( a[i] \) thanks to the increment of 2.

- for (int i = 1; i < LEN; i += 1)
  \[ a[i] = a[i - 1] + b[i]; \]

- True dependency:
  - Processing element \( i \) requires the results of the previous iteration
VECTORISATION: Compilers vectorisation efficiency

- Icc average vectorisation %: 43.1%
- Armclang average vectorisation %: 50.0%
VECTORISATION: Induction variable recognition

- Induction variables are modified and/or defined in the loop
- A typical example:
  - for (int i = 0; i < LEN - 1; i++) {
    j = i + 1;
    a[j] = a[j] + b[i];
  }
- False dependency:
  - j is set in every iteration but is always i + 1
VECTORISATION: Induction variable recognition

- Icc average vectorisation %: 56.9%
- Armclang average vectorisation %: 28.6%
The compiler must be able to interchange loops to perform vectorization

A typical example:

```c
for (int i = 0; i < LEN ; i++) {
    for (int j = 1; j < LEN ; j++) {
        a[j][i] = a[j][i] + b[j][i];
    }
}
```

False dependency:
- Without permuting loops, expression cannot be vectorized
VECTORISATION: Loop Interchange

- Icc average vectorisation %: 50.0%
- Armclang average vectorisation %: 0%
- LLVM's loop interchange flag not recognized
VECTORISATION: Control Flow

- If statement inside the loop
- A typical example:
  ```
  for (int i = 0; i < LEN - 1; i++) {
    if (b[i] > (float)0.)
      a[i] = a[i] * b[i];
  }
  ```
  - The compiler must handle the singularity to be able to vectorise
VECTORISATION: Control Flow

- Icc average vectorisation %: 92.7%
- Armclang average vectorisation %: 0.9%
- Vectorisation report show issues with control flow
MAQAO for AArch64 architectures

- MAQAO for AArch64 is released:
  - Guide users in their optimisation process
  - Evaluate compilers output according to your micro-architecture

- Armclang vectorisation analysis:
  - Missing loop interchange flag
  - Room for improvement in control flow handling
Thank you for your attention!

Questions?