HPC Compiler Optimisations

Masaki Arai
Takahiro Miyoshi
Renato Golin
Outline

● Overview of LLVM for HPC
● Activating software pipelining for AArch64
● Eliminating redundant branches for HPC applications
● Improving register allocations for HPC applications
● Vectorisation/SIMDisation
● Evaluating the quality of compilers continually
● Future work
Overview of LLVM for HPC

- We are enhancing LLVM for HPC applications
- Many optimisations and patches are currently being introduced into LLVM for AArch64
- Significant optimisation flow for HPC applications
  1. Pre-processing code for HPC kernels
  2. Vectorisation/SIMDisation considering SVE
  3. Software pipelining considering SVE
  4. Allocating registers for HPC kernels
  5. Distributing huge and/or complex loops
  6. Distributing loops considering hardware resources

Today’s presentation will explain our activities on 1 to 4
Activating software pipelining(1)

- MachinePipeliner was introduced from LLVM 4.0
  - It implements Swing Modulo Scheduling algorithm
- However, its target architecture is currently Hexagon family CPU only
- We are porting it to AArch64 by the following works
  - Extending SMSchedule for multiple CPU scheduling models of LLVM
  - Adding code specific to AArch64 for recognizing loop induction variables and conditional branches
- We succeeded in generating software pipelined code for AArch64
Activating software pipelining (2)

- However, there is still much work to do
  - Extending LLVM's CPU scheduling model for MachinePipelinier
  - Increasing loop patterns that can be optimized
  - Supporting data dependence distance of iterations greater than 1
  - Generating code by making effective use of SVE
Eliminating redundant branches for HPC applications (1)

- Reported problems of LLVM on HKG18

```c
void calculateNewPfMTwo_child(...) {
 ...
  for (msi = 0; msi < nsize; msi++) {
      ...
  }
 ...
  for (msi = 0; msi < nsize; msi++) {
      ...
  }
  for (msi = 0; msi < nsize; msi++) {
      ...
      for (msj = 0; msj < nsize; msj++) {
      }
  }  
 ...
  return;  
}  
```
Eliminating redundant branches for HPC applications(2)

- Outline of optimisation effect

After global value numbering, some basic blocks share %cmp
Eliminating redundant branches for HPC applications (3)

- Optimisation for reported problems

Our patch doesn’t duplicate loop headers, big basic blocks, or basic blocks with inline asm
Eliminating redundant branches for HPC applications(4)

- Effectiveness of this improvement for SPEC CPU 2017

<table>
<thead>
<tr>
<th>benchmark</th>
<th>#F</th>
<th>#BB</th>
<th>#UC</th>
<th>#DUP</th>
</tr>
</thead>
<tbody>
<tr>
<td>600.perlbench_s</td>
<td>256</td>
<td>1106</td>
<td>797</td>
<td>309</td>
</tr>
<tr>
<td>602.gcc_s</td>
<td>1401</td>
<td>5293</td>
<td>3920</td>
<td>1373</td>
</tr>
<tr>
<td>605.mcf_s</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>619.lbm_s</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>620.omnetpp_s</td>
<td>78</td>
<td>340</td>
<td>136</td>
<td>204</td>
</tr>
<tr>
<td>623.xalancbmk_s</td>
<td>193</td>
<td>464</td>
<td>346</td>
<td>118</td>
</tr>
<tr>
<td>625.x264_s</td>
<td>129</td>
<td>376</td>
<td>238</td>
<td>138</td>
</tr>
<tr>
<td>631.deepsjeng_s</td>
<td>9</td>
<td>42</td>
<td>35</td>
<td>7</td>
</tr>
<tr>
<td>638.imagick_s</td>
<td>115</td>
<td>542</td>
<td>278</td>
<td>264</td>
</tr>
<tr>
<td>641.leela_s</td>
<td>14</td>
<td>31</td>
<td>7</td>
<td>24</td>
</tr>
<tr>
<td>644.nab_s</td>
<td>34</td>
<td>93</td>
<td>52</td>
<td>41</td>
</tr>
<tr>
<td>657.xz_s</td>
<td>17</td>
<td>30</td>
<td>18</td>
<td>12</td>
</tr>
</tbody>
</table>

We added this improvement into JumpThreading

#F: the number of function in which our patch works
#UB: the number of basic block which our patch can convert into unconditional
#DUP: the number of basic block which our patch can convert into unconditional after duplicating them
Improving register allocation for HPC applications

- We reported problems of Greedy Register Allocator of LLVM
  - It doesn’t handle well HPC kernel loops
- Our current trial implementations for its improvement
  1. Changing generated COPY instructions’ order when eliminating PHI nodes for register allocation
  2. Changing hardware register orders in `AllocationOrder`
  3. Using ‘hint’ information smarter
  4. Using new split-or-spill strategy based on HPC loop structures

```
AllocationOrder(FPR32) =
[ %s0 %s1 %s2 %s3 %s4 %s5 %s6 %s7 %s16 %s17 %s18 %s19 %s20
 %s21 %s22 %s23 %s24 %s25 %s26 %s27 %s28 %s29 %s30 %s31
 %s8 %s9 %s10 %s11 %s12 %s13 %s14 %s15 ]
```

%2 = COPY %1
Assigning %2 to %s1

Hints for %1: %s1
Vectorisation/SIMDisation

- Vectorisation is a significant optimisation that affects the performance of AArch64+SVE
  - Vectorisation is realized by various functions of the compiler
    - Using special function or library function
    - Using directives like OpenMP
    - Using Polly based on integrated loop optimisation
    - Using VPlan which estimates effects by the cost model
    - By instruction pattern matching
    - By target machine specific code generation

- One of our activities is to add an instruction pattern to LoopUtils
  - This makes the s3111 in TSVC benchmark 2.5 times faster on Cavium ThunderX2

```c
sum = 0.;
for (int i = 0; i < LEN; i++) {
    if (a[i] > (float)0.) {
        sum += a[i];
    }
}
```
Evaluating the quality of compilers continually

- We have created tools to support compiler quality checking
  - HPC Compiler Quality Checker
    - https://github.com/Linaro/hcqc
- We want to track changes in the quality of the compiler automatically
- We have added the following features to this tool
  - Increasing execution time and cache miss
  - Increasing spill code
  - Decreasing instruction level parallelism
  - Not generating important instructions
  - Increasing complexity of control flow graph
  - ...

Increasing the score ➔ Alarm!
Evaluating the quality of compilers continually(2)

- We have acquired C and D data
- The question is how to determine the value of the weight W
- The difficult problem is how to evaluate significant changes in the control flow graph
  - Loop parallelization, loop tiling, loop unrolling, etc.
- However, there are probably no proper methods for these
- We want to keep this tool practical and simple!

\[
E_{\text{score}} = E_{\text{perf}} + E_{\text{cfg}} + E_{\text{op}} + E_{\text{height}} + E_{\text{regalloc}} \quad (1)
\]
\[
E_{\text{perf}} = W_{0,1} C_{\text{time}} + \sum_{x \in \text{CACHE}} W_{0,x} C_{0,x} \quad (2)
\]
\[
E_{\text{cfg}} = \sum_{x \in \text{BB}} W_{1,x} D_{1,x} C_{1,x} \quad (3)
\]
\[
E_{\text{op}} = \sum_{x \in \text{OP}} W_{2,x} D_{2,x} C_{2,x} \quad (4)
\]
\[
E_{\text{height}} = \sum_{x \in \text{BB}} W_{3,x} D_{3,x} C_{3,x} \quad (5)
\]
\[
E_{\text{regalloc}} = \sum_{x \in \text{SPILL}} W_{4,x} D_{4,x} C_{4,x} \quad (6)
\]

C: collected data from dynamic and static results
D: the depth of the control flow graph associated with C
W: the weight for each term
E: the score for each metric
Future Work

- Pushing out patches to LLVM upstream
- Developing the optimisation flow for HPC applications
  1. Pre-processing code for HPC kernels
  2. Vectorisation/SIMDisation considering SVE
  3. Software pipelining considering SVE
  4. Allocating registers for HPC kernels
  5. **Distributing huge and/or complex loops**
  6. Distributing loops considering hardware resources
- Evaluating the quality of compilers continually