Software Optimization for a RISC-V Accelerator

A case study

Julien de Castelnau

There's Plenty of Room at the Top... And Plenty of Work Too

Version	Implementation	Running time (s)	GFLOPS	Absolute speedup	Relative speedup	Fraction of peak (%)
1	Python	25,552.48	0.005	1	–	0.00
2	Java	2,372.68	0.058	11	10.8	0.01
3	C	542.67	0.253	47	4.4	0.03
4	Parallel loops	69.80	1.969	366	7.8	0.24
5	Parallel divide and conquer	3.80	36.180	6,727	18.4	4.33
6	plus vectorization	1.10	124.914	23,224	3.5	14.96
7	plus AVX intrinsics	0.41	337.812	62,806	2.7	40.45

- ≈1300x speedup between C versions

There's Plenty of Room at the Top... And Plenty of Work Too

Version	Implementation	Running time (s)	GFLOPS	Absolute speedup	Relative speedup	Fraction of peak (%)
1	Python	25,552.48	0.005	1	–	0.00
2	Java	2,372.68	0.058	11	10.8	0.01
3	C	542.67	0.253	47	4.4	0.03
4	Parallel loops	69.80	1.969	366	7.8	0.24
5	Parallel divide and conquer	3.80	36.180	6,727	18.4	4.33
6	plus vectorization	1.10	124.914	23,224	3.5	14.96
7	plus AVX intrinsics	0.41	337.812	62,806	2.7	40.45

- ≈1300x speedup between C versions
- Have to repeat for every new HW platform!
- How do we write software like this for custom hardware?

Case Study

- Hardware: Embedded RISC-V CPU with dense 4x4 matrix multiplication accelerator
- Programming model:
  - Performs operations on matrices w/ CPU instructions (RISC-V ISE)
  - Matrices stored in special registers $m_0 \dots m_7$
  - E.g. `mmasa.w m2,m0,m1` computes $m_2 \mathrel{+}= m_0 \cdot m_1^T$
- Software: simplified 1D convolution layer of a CNN (Convolutional Neural Network)
- Manual optimization process: demo!

Background

- 1D convolution: Given an input stream $I[I_C][N]$ of length N with I_C input channels, and a array of kernels $K[O_C][I_C][W]$ of width W with O_C output channels, compute output $O[O_C][N]$ s.t.

$$O[i][j] = \sum_{c=0}^{I_C} \sum_{r=0}^{W} \begin{cases} I[c][j+r] \cdot K[i][c][r] & \text{if } j+r < N \\ 0 & \text{otherwise} \end{cases}, 0 \leq i \leq O_C, 0 \leq j \leq N$$

Background

- 1D convolution: Given an input stream $I[I_C][N]$ of length N with I_C input channels, and a array of kernels $K[O_C][I_C][W]$ of width W with O_C output channels, compute output $O[O_C][N]$ s.t.

$$O[i][j] = \sum_{c=0}^{I_C} \sum_{r=0}^{W} \begin{cases} I[c][j+r] \cdot K[i][c][r] & \text{if } j+r < N \\ 0 & \text{otherwise} \end{cases}, 0 \leq i \leq O_C, 0 \leq j \leq N$$

- Tiled matrix multiplication:

$$X = \begin{bmatrix} x_{0,0} & x_{0,1} & x_{0,2} & x_{0,3} \\ x_{1,0} & x_{1,1} & x_{1,2} & x_{1,3} \\ x_{2,0} & x_{2,1} & x_{2,2} & x_{2,3} \\ x_{3,0} & x_{3,1} & x_{3,2} & x_{3,3} \end{bmatrix}, Y = \begin{bmatrix} y_{0,0} & y_{0,1} & y_{0,2} & y_{0,3} \\ y_{1,0} & y_{1,1} & y_{1,2} & y_{1,3} \\ y_{2,0} & y_{2,1} & y_{2,2} & y_{2,3} \\ y_{3,0} & y_{3,1} & y_{3,2} & y_{3,3} \end{bmatrix}$$

$$\implies X \cdot Y = \begin{bmatrix} X_{0,0}Y_{0,0} + X_{0,1}Y_{1,0} & X_{0,0}Y_{0,1} + X_{0,1}Y_{1,1} \\ X_{1,0}Y_{0,0} + X_{1,1}Y_{1,0} & X_{1,0}Y_{0,1} + X_{1,1}Y_{1,1} \end{bmatrix}$$

Summary

- Performance 16x, but *code length* 5x!
- Not shown: time to develop, debug, discard bad ideas..
- No longer immediately evident what the program does
- Problems worsen as source problem becomes more complex

Summary

- Performance 16x, but *code length* 5x!
- Not shown: time to develop, debug, discard bad ideas..
- No longer immediately evident what the program does
- Problems worsen as source problem becomes more complex
- What if we captured our incremental *refinement* of the code, as code itself?

User-schedulable languages

- Idea: Separate the algorithm from the way it is carried out - the *schedule*

User-schedulable languages

- Idea: Separate the algorithm from the way it is carried out - the *schedule*
- Programmer optimizes by writing schedule, while preserving semantics of algorithm
- Schedule expresssed in code as *incremental rewrites* of algorithm

User-schedulable languages

- Idea: Separate the algorithm from the way it is carried out - the *schedule*
- Programmer optimizes by writing schedule, while preserving semantics of algorithm
- Schedule expresssed in code as *incremental rewrites* of algorithm
- Many USLs proposed - not a complete literature survey!

TVM

- End-to-end deep learning compiler with user-schedulable component
- Write algorithm in Python-embedded DSL, e.g. matmul:

```
@T.prim_func
def matmul(A: T.Buffer((N, N), "float32"),
    B: T.Buffer((N, N),  "float32"),
    C: T.Buffer((N, N), "float32")) -> None:
    for i,j,k in T.grid(N, N, N):
        vi, vj, vk = T.axis.remap("SSR", [i,j,k])
        C[vi, vj] = C[vi, vj] + A[vi, vk] * B[vj, vk]
```

- Use Python functions to rewrite programs, forming schedule
  - e.g. `reorder_loops()` reorders nested loops, `split()` does tiling
- TVM compiler generates C output, fed to LLVM/GCC

TVM limitations

- Can't represent HW assembly instructions in TVM IR

TVM limitations

- Can't represent HW assembly instructions in TVM IR
- Two options:
  1. Write an LLVM backend to perform instruction selection
  2. Handwrite an optimized "microkernel" abstracting away accelerator details
- Both invisible to schedule: Either forced to defer to compiler or blackbox subroutine

TVM limitations

- Can't represent HW assembly instructions in TVM IR
- Two options:
  1. Write an LLVM backend to perform instruction selection
  2. Handwrite an optimized "microkernel" abstracting away accelerator details
- Both invisible to schedule: Either forced to defer to compiler or blackbox subroutine
- Performance engineer: "Why should I write a compiler? Why should I use TVM if I have to handwrite C anyway?"

TVM limitations

- Can't represent HW assembly instructions in TVM IR
- Two options:
  1. Write an LLVM backend to perform instruction selection
  2. Handwrite an optimized "microkernel" abstracting away accelerator details
- Both invisible to schedule: Either forced to defer to compiler or blackbox subroutine
- Performance engineer: "Why should I write a compiler? Why should I use TVM if I have to handwrite C anyway?"
- Can we make this code generation a scheduling decision?

Exo

- Python DSL compiled to C like TVM, but HW support is *externalized* to user, instead of compiler
- Provide *instruction* specification and implementation
- Backed by a formal model $\implies$ Every re-write is a correct Exo program
- Demo!

OptiTrust

- Rewrite C/C++ sources directly using OCaml scripts
- Transformations formally backed by separation logic
- Supports most of the C spec $\implies$ no artificial limitations due to language?
- Demo!

OptiTrust & Exo

- DSL often insufficient (TVM & Exo), but even the C spec is not enough! (OptiTrust)
- Challenging language design problem:
  - Need inherently nonstandard custom HW-specific constructs
  - Expect powerful static analysis to ensure correctness
  - But also want full expressiveness of C language
  - While being simple enough to warrant learning!

Conclusion

- Modern high-performance code:
  1. is time-consuming to write, debug, maintain
  2. must employ every optimization available
  3. necessarily demands a high degree of control over HW, down to assembly
- User-scheduled languages help (1), but repeatedly suffer from (2) and (3)!
  - Due to (2), when they fail, or are too cumbersome, they go out the window!

Conclusion

- Modern high-performance code:
  1. is time-consuming to write, debug, maintain
  2. must employ every optimization available
  3. necessarily demands a high degree of control over HW, down to assembly
- User-scheduled languages help (1), but repeatedly suffer from (2) and (3)!
  - Due to (2), when they fail, or are too cumbersome, they go out the window!

USLs must contend with challenging design requirements to be considered for replacing industry-standard handwritten C,C++

Thank you!