atJIT: a just-in-time autotuning compiler for C++

Kavon Farvardin • 23.07.2018

Host: Hal Finkel

Just-in-time Compilation

Not a new idea:

1. Provide access to the compiler *while* executing the program.

2. The executable (re)compiles part of itself on-demand.

Why?

Runtime information can significantly improve the quality of compiled code.

For computations like those at Argonne, this is very useful.

```
int slowMul (int a, int b) {
  int iters = b, answer = 0;
  if (b < 0) {
    // if b == -1*c
    //   then a * b == (a * -1) * c
    a = -a;
    iters = -b;
  }
  for (; iters > 0; --iters)
    answer += a;
  return answer;
}
```

An implementation of integer multiplication that only uses addition & negation.

```
define i32 @slowMul(i32, i32) {
  %3 = icmp slt i32 %1, 0
  %4 = sub nsw i32 0, %1
  %5 = select i1 %3, i32 %4, i32 %1
  %6 = icmp sgt i32 %5, 0
  br i1 %6, label %7, label %16

; <label>:7:
  %8 = sub nsw i32 0, %0
  %9 = select i1 %3, i32 %8, i32 %0
  %10 = xor i32 %5, -1
  %11 = icmp sgt i32 %10, -2
  %12 = select i1 %11, i32 %10, i32 -2
  %13 = add i32 %5, %12
  %14 = add i32 %13, 2
  %15 = mul i32 %9, %14
  br label %16

; <label>:16:
  %17 = phi i32 [ 0, %2 ], [ %15, %7 ]
  ret i32 %17
}
```

A regular compiler can manage to optimize out the loop, but it’s still inefficient!

```
const int b = // some dynamic value
for (int a = 0; a < 9999999; ++a) {
  ... = slowMul(a, b);
  ...
```

Let’s dynamically compile a specialized version!

At start of loop,
we observed b = 3

```
define i32 @slowMul(i32) {
  %2 = mul i32 %0, 3
  ret i32 %2
}
```

We got lucky and produced the best possible version!

```
-faggressive-loop-optimizations  -falign-functions=<...>  -falign-jumps=<...>  -falign-labels=<...>  -falign-loops=<...>  -fassociative-math
-fauto-profile  -fauto-profile[=path]  -fauto-inc-dec  -fbranch-probabilities  -fbranch-target-load-optimize  -fbranch-target-load-optimize2
-fbtr-bb-exclusive  -fcaller-saves  -fcombine-stack-adjustments  -fconserve-stack  -fcompare-elim  -fcprop-registers  -fcrossjumping  -fcse-follow-jumps
-fcse-skip-blocks  -fcx-fortran-rules  -fcx-limited-range  -fdata-sections  -fdce  -fdelayed-branch  -fdelete-null-pointer-checks  -fdevirtualize
-fdevirtualize-speculatively  -fdevirtualize-at-ltrans  -fdse  -fearly-inlining  -fipa-sra  -fexpensive-optimizations  -ffat-lto-objects  -ffast-math
-ffinite-math-only  -ffloat-store  -fexcess-precision=style  -fforward-propagate  -ffp-contract=style  -ffunction-sections  -fgcse  -fgcse-after-reload
-fgcse-las  -fgcse-lm  -fgraphite-identity  -fgcse-sm  -fhoist-adjacent-loads  -fif-conversion  -fif-conversion2  -findirect-inlining
-finline-functions  -finline-functions-called-once  -finline-limit=n  -finline-small-functions  -fipa-cp  -fipa-cp-clone  -fipa-bit-cp -fipa-vrp
-fipa-pta  -fipa-profile  -fipa-pure-const  -fipa-reference  -fipa-icf  -fira-algorithm=algorithm  -fira-region=region  -fira-hoist-pressure
-fira-loop-pressure  -fno-ira-share-save-slots  -fno-ira-share-spill-slots  -fisolate-erroneous-paths-dereference  -fisolate-erroneous-paths-attribute
-fivopts  -fkeep-inline-functions  -fkeep-static-functions  -fkeep-static-consts  -flimit-function-alignment  -flive-range-shrinkage  -floop-block
-floop-interchange  -floop-strip-mine  -floop-unroll-and-jam  -floop-nest-optimize  -floop-parallelize-all  -flra-remat  -flto  -flto-compression-level
-flto-partition=alg  -fmerge-all-constants  -fmerge-constants  -fmodulo-sched  -fmodulo-sched-allow-regmoves  -fmove-loop-invariants
-fno-branch-count-reg  -fno-defer-pop  -fno-fp-int-builtin-inexact  -fno-function-cse  -fno-guess-branch-probability  -fno-inline  -fno-math-errno
-fno-peephole  -fno-peephole2  -fno-printf-return-value  -fno-sched-interblock  -fno-sched-spec  -fno-signed-zeros  -fno-toplevel-reorder
-fno-trapping-math  -fno-zero-initialized-in-bss  -fomit-frame-pointer  -foptimize-sibling-calls  -fpartial-inlining  -fpeel-loops
-fpredictive-commoning  -fprefetch-loop-arrays  -fprofile-correction  -fprofile-use  -fprofile-use=path  -fprofile-values  -fprofile-reorder-functions
-freciprocal-math  -free  -frename-registers  -freorder-blocks  -freorder-blocks-algorithm=algorithm  -freorder-blocks-and-partition
-freorder-functions  -frerun-cse-after-loop  -freschedule-modulo-scheduled-loops  -frounding-math  -fsave-optimization-record  -fsched2-use-superblocks
-fsched-pressure  -fsched-spec-load  -fsched-spec-load-dangerous  -fsched-stalled-insns-dep[=n]  -fsched-stalled-insns[=n]  -fsched-group-heuristic
-fsched-critical-path-heuristic  -fsched-spec-insn-heuristic  -fsched-rank-heuristic  -fsched-last-insn-heuristic  -fsched-dep-count-heuristic
-fschedule-fusion  -fschedule-insns  -fschedule-insns2  -fsection-anchors  -fselective-scheduling  -fselective-scheduling2  -fsel-sched-pipelining
-fsel-sched-pipelining-outer-loops  -fsemantic-interposition  -fshrink-wrap  -fshrink-wrap-separate  -fsignaling-nans  -fsingle-precision-constant
-fsplit-ivs-in-unroller  -fsplit-loops -fsplit-paths  -fsplit-wide-types  -fssa-backprop  -fssa-phiopt  -fstdarg-opt  -fstore-merging  -fstrict-aliasing
-fthread-jumps  -ftracer  -ftree-bit-ccp  -ftree-builtin-call-dce  -ftree-ccp  -ftree-ch  -ftree-coalesce-vars  -ftree-copy-prop  -ftree-dce
-ftree-dominator-opts  -ftree-dse  -ftree-forwprop  -ftree-fre  -fcode-hoisting  -ftree-loop-if-convert  -ftree-loop-im  -ftree-phiprop
-ftree-loop-distribution  -ftree-loop-distribute-patterns  -ftree-loop-ivcanon  -ftree-loop-linear  -ftree-loop-optimize  -ftree-loop-vectorize
-ftree-parallelize-loops=n  -ftree-pre  -ftree-partial-pre  -ftree-pta  -ftree-reassoc  -ftree-sink  -ftree-slsr  -ftree-sra  -ftree-switch-conversion
-ftree-tail-merge  -ftree-ter  -ftree-vectorize  -ftree-vrp  -funconstrained-commons  -funit-at-a-time  -funroll-all-loops  -funroll-loops
-funsafe-math-optimizations  -funswitch-loops  -fipa-ra  -fvariable-expansion-in-unroller  -fvect-cost-model  -fvpt  -fweb  -fwhole-program  -fwpa
-fuse-linker-plugin  --param name=value -O  -O0  -O1  -O2  -O3  -Os  -Ofast  -Og
```

GCC optimization options (https://gcc.gnu.org/onlinedocs/gcc/Option-Summary.html)

Online autotuning

Objective: Easy-to-use automatic performance tuning via JIT compilation.

atJIT serves as a vehicle for experimenting with autotuning techniques.

Primary “knobs” we are focussing on:

1. Compiler Optimizations
   a. Loop transforms
   b. Inlining
   c. Pass ordering/choices.
2. Algorithmic Choices

Example Kernel

```
// multiply square matrices
template <typename T>
T** MatMul(const int DIM, T** aMatrix, T** bMatrix) {
  T** product = calloc_mat<T>(DIM);
  for (int row = 0; row < DIM; row++) {
    for (int col = 0; col < DIM; col++) {
      for (int inner = 0; inner < DIM; inner++) {
        product[row][col] += aMatrix[row][inner] * bMatrix[inner][col];
      }
    }
  }
  return product;
}
```

Programmer's Interface

```
auto const &OptimizedFun = AT.reoptimize(MatMul<ElmTy>,
      DIM, _1, _2,  // arg 1 is fixed to DIM
      tuner_kind(AT_Random));

ElmTy** ans = OptimizedFun(aMatrix, bMatrix);
```

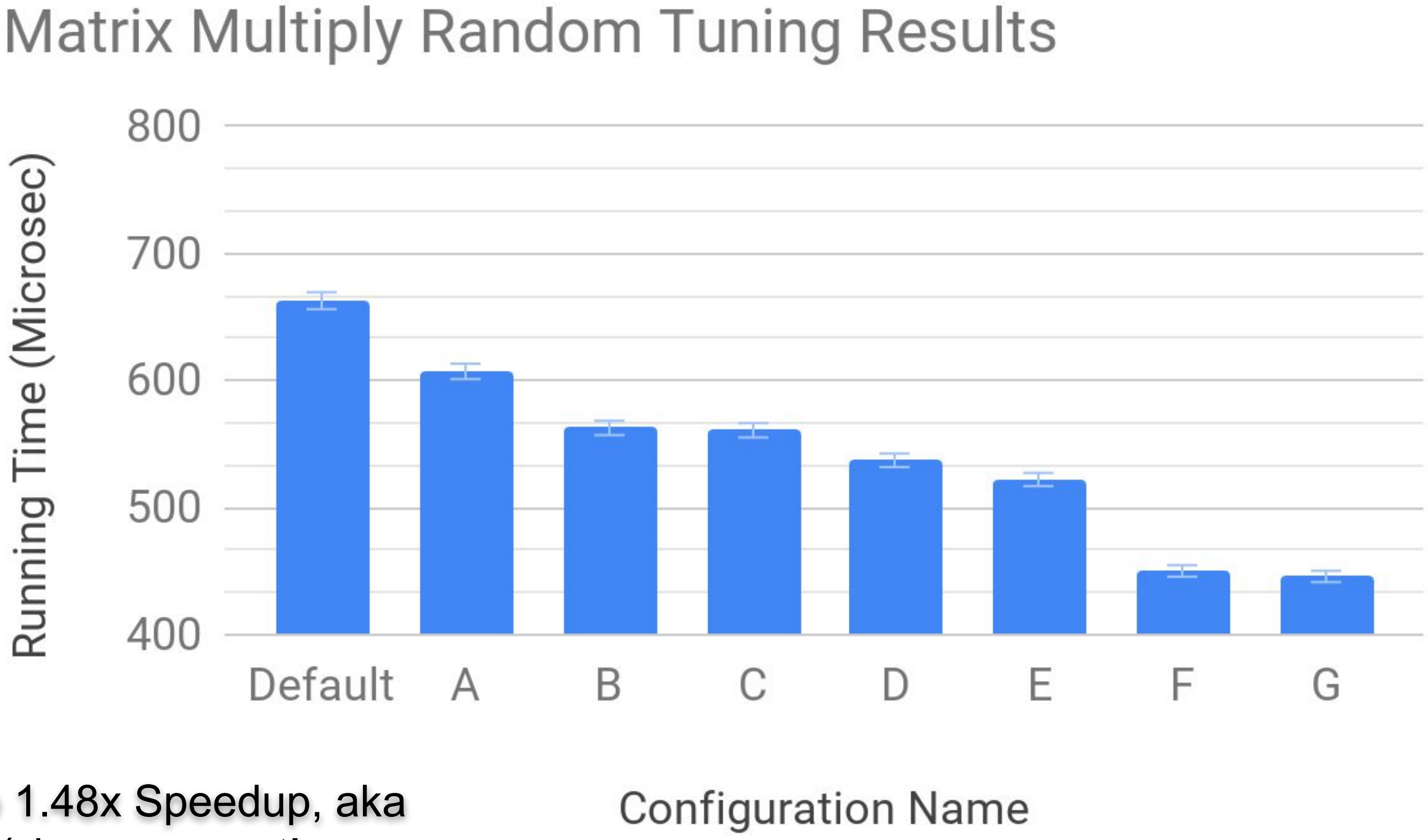
Compile with a simple, drop-in replacement of of CXX:

```
➤ ../install/bin/easycc -O2 matmul.cpp -o matmul
```

Automatic Tuning Process

```
avg time: 662929 ns, error: 0.401664%, (s^2: 3.10198e+07, s: 5954.09, SEM: 2662.75) from 5 measurements
{
inlining threshold := 225
loop #3 := <>
loop #2 := <>
loop #0 := <>
loop #1 := <>
}

avg time: 445898 ns, error: 0.998717%, (s^2: 2.54827e+09, s: 50579.3, SEM: 4453.26) from 129 measurements
{
inlining threshold := 976
loop #3 := <llvm.loop.unroll.full: 1, llvm.loop.licm_versioning.disable: 0, llvm.loop.distribute.enable: 1, >
loop #2 := <llvm.loop.unroll.full: 1, llvm.loop.vectorize.enable: 1, llvm.loop.vectorize.width: 8, >
loop #0 := <llvm.loop.unroll.count: 32, llvm.loop.interleave.count: 1, llvm.loop.distribute.enable: 0, >
loop #1 := <llvm.loop.unroll.full: 1, llvm.loop.licm_versioning.disable: 1, llvm.loop.distribute.enable: 0, >
}
```



Up to 1.48x Speedup, aka
32.7% improvement!

Challenges

- Running-time normalization
- Parameter optimization
- Parallelizing JIT compilation
- … and certainly more!

```
void performWork (dataset, iterations) { ... }

void main () {
ATDriver AT;

auto func1 = AT.reoptimize(performWork, ...);
func1(dataset, 100);

auto func2 = AT.reoptimize(performWork, ...);
func2(dataset, 500);
...
```

Conclusion

- I’m still here for 1¼ more months. My TODOs include:
  - Bayesian Optimization
  - Simulated Annealing
  - Algorithmic Tuning

- Special thanks to
  - Hal Finkel & Michael Kruse (Argonne)
  - John Reppy (UChicago)
  - Serge Guelton & Juan Manuel Martinez Caamaño
    (developed open-source infrastructure we started from)

Code is available on GitHub:
https://github.com/**kavon/atJIT**