Advanced Systems Lab - FFT

8337 characters in 1374 words on 250 lines

Florian Moser

July 8, 2020

1 fast fourier transform (FFT)

1.1 introduction

history
1805 Gauss discovered FFT for personal paper-interpolation
signal processing done analog (with currents)
1965 FFT cooley tukey rediscovers; enables digital processing

purpose
change of basis by multiplying with fixed matrix
is a linear transform (n-vector input/output)
y = Tx (y_k = sum_of(t_kl *x_l))

optimization potential
up to 35x faster
5x for locality, 3x for vectorization, 3x for threading

1.2 discrete fourier transform (DFT)

y_k = sum_of(e^(-2*kl*pi*i / n) *x_l)
all n roots of unity (of one)
w_n called primitive nth root of 1

notation
y = DFT_n*x, $2n^2$ operations

evaluate
get multiplicator using n (size) m = kl*2pi/n
insert k/l (row/column; zero-based)
then evaluate with cos(a) + i*sin(m)
remember that sin(0) = 0, cos(0) = 1

DFT_2
[[1 1], [1 -1]]
no mults (bc all factors 1), 1 adds/sub
$y_1 = x_1$ + x_2; $y_2 = x_1$ - x_2
called butterfly

DFT_4
[1 1 1 1],[1 i -1 -i],[1 -1 1 -1],[1 -i -1 i]
12 adds, 4 mults (if not reduced)
in complex arithmetic, needs to be mapped to machine

further examples
DFT & RDFT universally used
MPEG & JPEG uses own kind of DFT

1.3 cooley-tukey FFT

FFT transformation algorithm
calculate T like T = T_1*T_2*.... $*T_m$
only useful if T_i are sparse, and m low

1.3.1 FFT=4

essentially 4 DFT of size 2

matrix
for . = 0 (better readability)
T1 = [1...,..1.,.1..,...1] (shift; 0 ops)
T2 = [1 1..,1 -1..,..1 1,..1 -1] (four adds)
T3 = [1...,.1..,..1.,...i] (one mul by i)
T4 = [1.1.,.1.1,1.-1.,.1.-1] (four adds)
y = T4*T3*T2*T1*x
⇒ reduced FFT opcount to 8 adds, 1 mul

algebra
(DFT_2 (x) I_2)diag(1,1,1,i)(I_2 (x) DFT_2)L_2^4
I_2 = diag(1,1) is identity matrix of size 2
L_2^4 = permutation with stride 2
for (x) kroenecker product
multiply entry on left with whole matrix right
produces for 2*2 (x) n*n = 2n*2n result

dataflow representation
right x1,x2,x3,x4 flows to left y1y2y3y4
permutation of x2 and x3 (L_2^4)
DFT_2 applied to x1,x2 and x3x4 (I_2 (x) DFT_2)
scaling represented as black dots (diag(1,1,1,i))
DFT_2 applied to x1,x3 and x2x4 (DFT_2 (x) I_2)

1.3.2 generalization

(DFT_k (x) I_m)T_m^n(I_k (x) DFT_m)L_k^n
choose radix k, select m to fit input
DFT_k (x) (x) I_m results with DFT_k in the diagonal
T_m^n as diagonal with m-1 1s, then an i, then repeat
I_k (x) DFT_m results in m A's at stride m
L_k^n permutation; read at stride k, write at stride 1
can reformulate to read at stride 1, write at stride m
get n logn algorithm

variants
decimation-in-time
decimation-in-frequency (transposed cooley-turkey)

factors
can choose radix k, but in the end k*m = n
for factors of 2 trivial; for primes need different algorithm

cost
n log(n) complex adds; each needs 2 real adds
n log(n)/2 complex mults; each needs 2 adds, 4 mults
hence 3n log(n) adds, 2n log(n) mults = 5n log(n)
reduce op count depending on recursion
because can simplify *i or *1

recursive vs iterative
calculation order different (DAG equivalent)
iterative in stages (compute all of same kind of butterfly)
recursive in recursion (compute butterfly as soon as dependency resolved)
only order of operation changes, not count
recursive better on caches bc temporal locality

1.4 FFT variants

iterative
initial permutation step
then bigger and bigger butterflies
size of butterflies defined by radix

pease
permute result of butterflies
to only operate on 2butterflies; good for CPUs

stockham
big butterflies and then permute
permutations always in pairs of two
good for 2-element cache lines

six-step FFT
optimized for parallelization
three stages of communication, two parallel stages
works for all sizes, any computation

multi-core FFT
similar to six-steps, but always two elements travel together
well if block size 2; no false sharing
hence two processes access elements in same cache line

1.5 complexity

use L_c as measure; defines cost add/mul = 1 for number <|c|
used L_2 in proofs; L_infinity would be more real-worldly

upper bounds
for n = 2^k ⇒ 3/2 n log(n)
other n ⇒ 8 n log(n)

lower bound
L_c(DFT) ≥ 1/2 n log_c(n)
hence as n goes to infinity then constant c also goes to infinity
fastest algorithm found is close under 4n log(n) for n=2ˆk
fastest algorithm in use needs little more than 4nlogn

1.6 optimizations

1.6.1 choice of algorithm

iterative used to be the best choice
bc easy to implement, fewer instructions
recursive now better
bc caches (temporal locality)

1.6.2 locality improvement

trivially need 4 data passes (bc 4 steps)

DFTrec
fuse T1,T2 (read stride k, DFT_m, write stride 1)
introduce interface DFTrec(m, x, y, k, j)
for m size, x/y vectors, k input stride, j output stride
out-of-place (reads x, writes y)
called recursively (just change stride)

DFTscaled
fuse T3,T4 (read stride m, prescale, DFT_k, write stride m)
introduce interface DFTscaled(k, y, d, m)
for k size, y input/output vector, d prescale, m input/output stride
in-place (reads & writes y)
base case (hence rewritten for each radix factor)

pseudocode
for n size, x/y input/output vector, t prescale lookup
choose k (depends on n; ensure base case for DFTscaled exists)
let m = n/k
for (i until k) DFTrec(m, x+i, y + m∗i, k, 1)
for (i until m) DFTscaled(k, y+i, t[i], m)

1.6.3 constants

FFT needs multiplications by roots of unity
expensive because needs sin/con to compute

precompute
DFT_init(n) for n input size to create lookup table
then reuse result for all same-sized DFT computations
prestore for small input sizes

1.6.4 optimized basic blocks

unroll recursions
bc hard for compiler to do it automatically
function calls overhead negligible (bc base case reached fast)

base cases
to optimize specifically for small inputs
empirically useful for n ≤ 32
but then need 31 DFTrec & DFTscaled each
can be generated (as its done by FFTW)

1.6.5 adaptivity

adapt recursion strategy to platform
do search within init function (reuse DFT_init(n))
use dynamic programming (saves perf. compared to exhaustive)

valid recursions
generated droplet for base cases must exist
DFTscaled as a basecase (hence always right-recursive)

1.7 FFTW (fastest fourier transform in the west)

generates codelets using SIMD
with input n generates DFTrec, DFTscaled in three steps

1.7.1 DAG generator

create trees based on sum formula
includes cooley-tukey, split-radix, ...
fuse trees into DAG

1.7.2 simplifier

simplify DAG

algebraic transformations
mults (0∗x ⇒ 0, similar cases for -1,1)
distribution laws (kx + ky ⇒ k(x+y))
canonicalization (x-y,y-x ⇒ x-y, -(y-x))

common subexpression elimination (CSE)
use temporary variables to avoid double computation

reduce constants (DFT specific)
has many multiplications with many different constants
each constant used positive & negative (bc sin(a) = -sin(a))
reduce register pressure by storing only positive constant
and use subtraction where -sin(a) would be needed

1.7.3 scheduler

transform DAG to c under register spill minimization
want to reach target I = O(log(C)) for cache size C
uses SSA style, scoping

approach
cut DAG in middle
then recurse on connected components (butterflies) to outside
hence use minimal number of registers for "durchstich"

cut DAG middle
start from the outsides (left = input, right = output)
color immediate next inner reachable nodes
repeat recursively until nodes touch in the middle

1.8 comparison

MMMM (ATLAS)
sparse MVM (Sparsity, Bebop)
DFT (FFTW)

cache optimizations
blocking for MMM, SMVM
recursive FFT & fusion of steps for FFTW

register optimizations
blocking for MMM, SMVM
schedule small FFT for FFT

optimized basic blocks
unrolling
scalar replacement & SSA
scheduling & simplifications for FFT

other optimizations
precompute constants for small DFT
for large FFT the algorithm becomes memory bound
hence avoid precomputation to save on storage

adaptivity
search blocking parameters for MMM
search register block size for SMVM
search recursion strategy for DFT

1.9 spiral

specifically to optimize FFT
for specific input size
applicable for locality, threading, ...

algorithm
use signal processing language (SPL)
is mathematical, declarative, point-free (any input size)
divide and conquer applied to transform algorithms

generate code
decompose DFT with SPL rules to algorithm
optimize for parallelization/vectorization
transform to structure with sums
optimize for locality
transform to C program
optimize in basic blocks

vectorization optimization
some SPL expressions directly relate to SIMD code
like DFT_2 (x) I_4 which can be transformed easily
hence rewrite given SPL to vectorizable SPL expressions

generate base cases
express intrinsics as matrices
then search for matrixes matching base cases

generalize for any input size
need many more codelets
results in huge library size
but faster than other approaches; including FFTW