Electrical & Computer ENGINEERING

How to Write Fast Code

18-645, spring 2008
19th Lecture, Mar. 26th

Instructor: Markus Püschel

TAs: Srinivas Chellappa (Vas) and Frédéric de Mesmay (Fred)

Summer Research Project

- **Preferred: undergraduate student**
- **Fulltime (40 hours/week), 3 months**
- **Pay: standard CMU (somewhere between 10 and 15/hour)**
- **Requirement: good standing in this class, overall GPA > 3.5**
- **Why?**
  - Research experience, maybe even publication
  - Good for grad school

Today

- **How to get a fast DFT: FFTW (version 2.x)**
  Focus on scalar code

- **References**
  - FFTW website
  - M. Frigo: A fast Fourier transform compiler

Optimizations

- Locality of data access (reuse)
- Precomputing constants
- Fast basic blocks
- Adaptivity

Optimizations

- Locality of data access (reuse)
  - Blackboard
- Precomputing constants
- Fast basic blocks
- Adaptivity

Optimizations

- Locality of data access (reuse)
- Precomputing constants
- Fast basic blocks
- Adaptivity

Precomputing Constants

- **The "twiddle" matrix T produces multiplications by constants that are sines and cosines:**

  `y[i] = sin(i·pi/128)·x[i]`

  Very expensive! (remember HW 2)

- **Solution:**
  - Precompute once and store in table
  - Reuse many times
  - Assumes transform is used many times (what if not?)

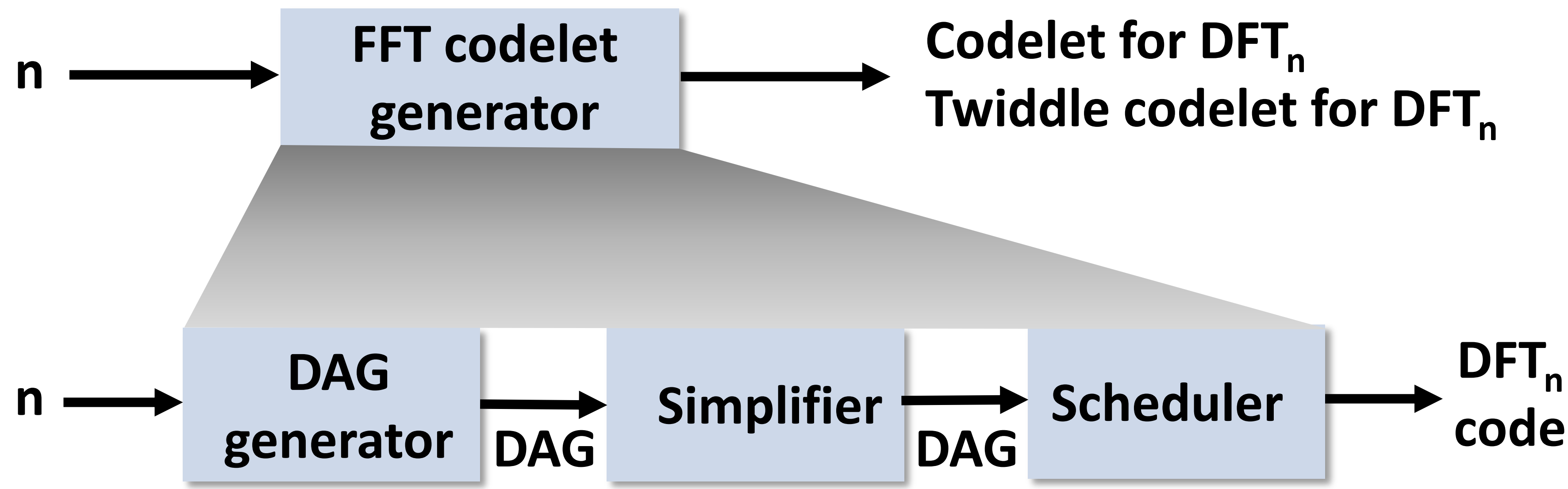
Optimizations

- **Locality of data access (reuse)**
- **Precomputing constants**
- **Fast basic blocks**
  - The FFTW codelet generator
- **Adaptivity**

Basic Block Optimizations for FFTs

- **Problem: similar to MMM**
  - We do not want to recurse all the way to n = 2
  - Infrastructure produces overhead = destroys performance.
- **Solution:**
  - Unrolled DFT code for fixed small sizes (≤ 32 say).
    In FFTW called codelets
- **Optimization for these blocks is much harder than for the micro MMMs in MMM**
- **Again, compilers often don't do a good job on unrolled code**
  - Doing it by hand you get a crisis (62 functions! Why 62?)
- **Solution: Code generator/optimizer for small sizes**

FFTW Codelet Generator



- **DAG: directed acyclic graph**
  - Represents a DFT algorithm (the dataflow)
  - Nodes: load, store, adds, mults by constant

- **Give example on blackboard**

DAG Generator

- **Knows FFTs: Cooley-Tukey, split-radix, Good-Thomas, Rader, represented in sum notation**

$$y_{n_2 j_1 + j_2} = \sum_{k_1=0}^{n_1-1} \left(\omega_n^{j_2 k_1}\right) \left(\sum_{k_2=0}^{n_2-1} x_{n_1 k_2 + k_1} \omega_{n_2}^{j_2 k_2} \right) \omega_{n_1}^{j_1 k_1}$$

- **For given n, suitable FFTs are recursively applied to yield n (real) expression trees for y_0, …, y_{n-1}**

- **Trees are fused to an (unoptimized) DAG**

Simplifier

- **Applies:**
  - algebraic transformations
  - common subexpression elimination (CSE)
  - DFT-specific optimizations
- **Algebraic transformations**
  - Simplify mults by 0, 1, -1
  - Distributivity law: kx + ky = k(x + y), kx + lx = (k + l)x
    May destroy common subexpressions and thus increase op count!
  - Canonicalization: (x-y), (y-x) to (x-y), -(x-y)
- **CSE: standard**
  - E.g., two occurrences of 2x+y: assign new temporary variable
- **DFT specific optimizations**
  - All numeric constants are made positive
  - Reason: constants need to be loaded into registers, too
  - CSE also on transposed DAG

Scheduler

- **Determines in which sequence the DAG is unparsed to C (topological sort of the DAG)**
  Goal: minimizer register spills

- **If R registers are available, then a 2-power FFT needs at least Ω(nlog(n)/R) register spills [1]**
  Same holds for a fully associative cache

- **FFTW's scheduler achieves this (asymptotic) bound independent of R**

- **Sketch it on blackboard**

[1] Hong and Kung: "I/O Complexity: The red-blue pebbling game"

Codelet Examples

- Notwiddle 2
- Notwiddle 3
- Twiddle 3
- Notwiddle 32

- **Techniques not seen before:**
  - Scoping (variables only defined where they occur)
    Purpose: simplifies dependency analysis
  - Single static assignment (SSA) style: Each variable has only one single definition in the code
    Purpose: no artificial dependencies

Optimizations

- **Locality of data access (reuse)**
- **Precomputing constants**
- **Fast basic blocks**
- **Adaptivity**
  - Start on blackboard

Dynamic Programming (DP)

- An algorithmic technique to solve optimization problems

- Definition: DP solves an optimization problem by caching and reusing subproblem solutions (memoization) rather than recomputing them

- Well-suited for all divide-and-conquer algorithms with a degree of freedom in the divide step

- Inherent assumption: Best solution is independent of the context in which the problem has to be solved

DP for FFTs

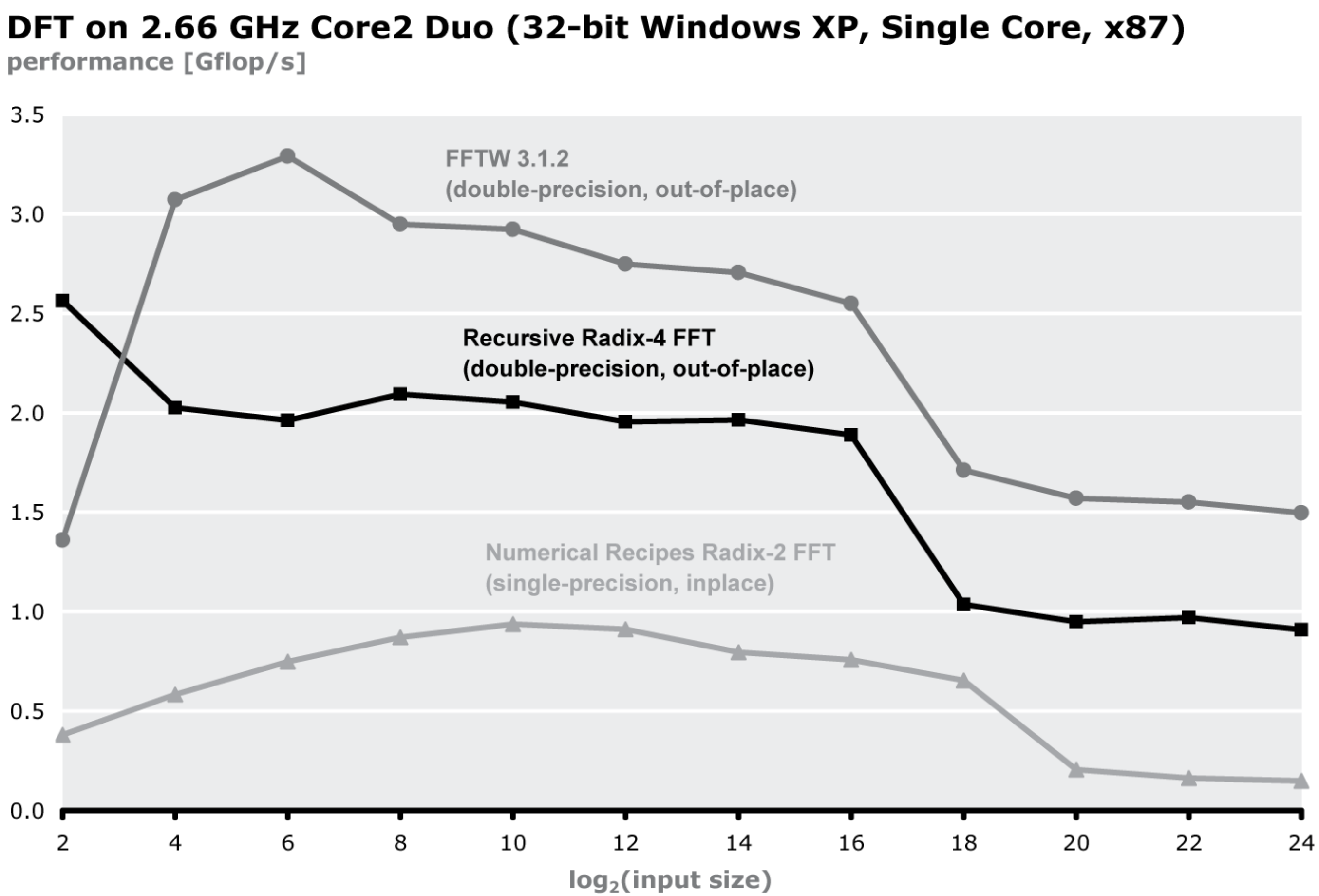
- Goal: Find the best recursion strategy for a DFT of size 2^k, computed with the Cooley-Tukey FFT
- Assume the best recursions for sizes $2^1,\dots,2^{k-1}$ are already computed
- Split DFT 2^k in all k-1 possible ways and use the best recursions for the smaller DFTs.
- The fastest of these k-1 algorithms is the solution for 2^k
- Cost: $(k-1)+(k-2)+\dots+1 = O(k^2)$ for size 2^k

DP for FFTs (cont'd)

- **In FFTW: Essentially as described on the previous slide, except left DFT is of size ≤ 64 (since twiddle codelet)**

- **Does DP assumption hold for FFTs?**
  - Not clear. In particular the best FFT could depend on the stride.
  - But works well in practice and is fast

Performance (Scalar Code)



The code for radix-4 FFT is in the tutorial

	MMM *Atlas*	**Sparse MVM** *Sparsity/Bebop*	**DFT** *FFTW*
Cache optimization	Blocking	Blocking (rarely useful)	recursive FFT, fusion of steps
Register optimization	Blocking	Blocking (sparse format)	Scheduling small FFTs
Optimized basic blocks	Unrolling, instruction ordering, scalar replacement, simplifications (for FFT)		
Other optimizations	—	—	Precomputation of constants
Adaptivity	Search: blocking parameters	Search: register blocking size	Search: recursion strategy