FFTW: AN ADAPTIVE SOFTWARE ARCHITECTURE FOR THE FFT

Matteo Frigo and Steven G. Johnson

Fan Kiat Chan
CS598APK

Motivation

- FFT literature has mostly focused on algorithms that minimize the number of floating-point operations
- On present-day computers, interactions with the processor pipeline and memory hierarchy have a larger impact on performance than number of floating-point operations
- Propose an adaptive FFT program that tunes the computation automatically for any particular hardware

Overview

- FFTW's main components: *executor, codelets* and *planner*.
- Executor (runtime)
  - Performs the computation of the transform by applying a *combination of codelets* specified by *planner*
- Codelets (compile time)
  - Specialized piece of code that computes part of the transform
  - *Generated automatically* during compile time using FFTW's codelet generator written in Caml Light
- Planner (runtime)
  - Determined during runtime before computation to construct a fast composition of codelets
  - Aims at *minimizing actual execution time* and not the number of floating point operations

Overview

```
fftw_plan plan;
COMPLEX A[n], B[n];

/* plan the computation */
plan = fftw_create_plan(n);

/* execute the plan */
fftw(plan, A);

/* the plan can be reused for
   other inputs of size N */
fftw(plan, B);
```

- User interacts with FFTW only through planner and executor
- Codelet generator is not used after compile time
  - user does not need to know Caml Light or need a Caml Light compiler
- FFTW creates a plan for a transform of a specified size and is reusable as many times as needed

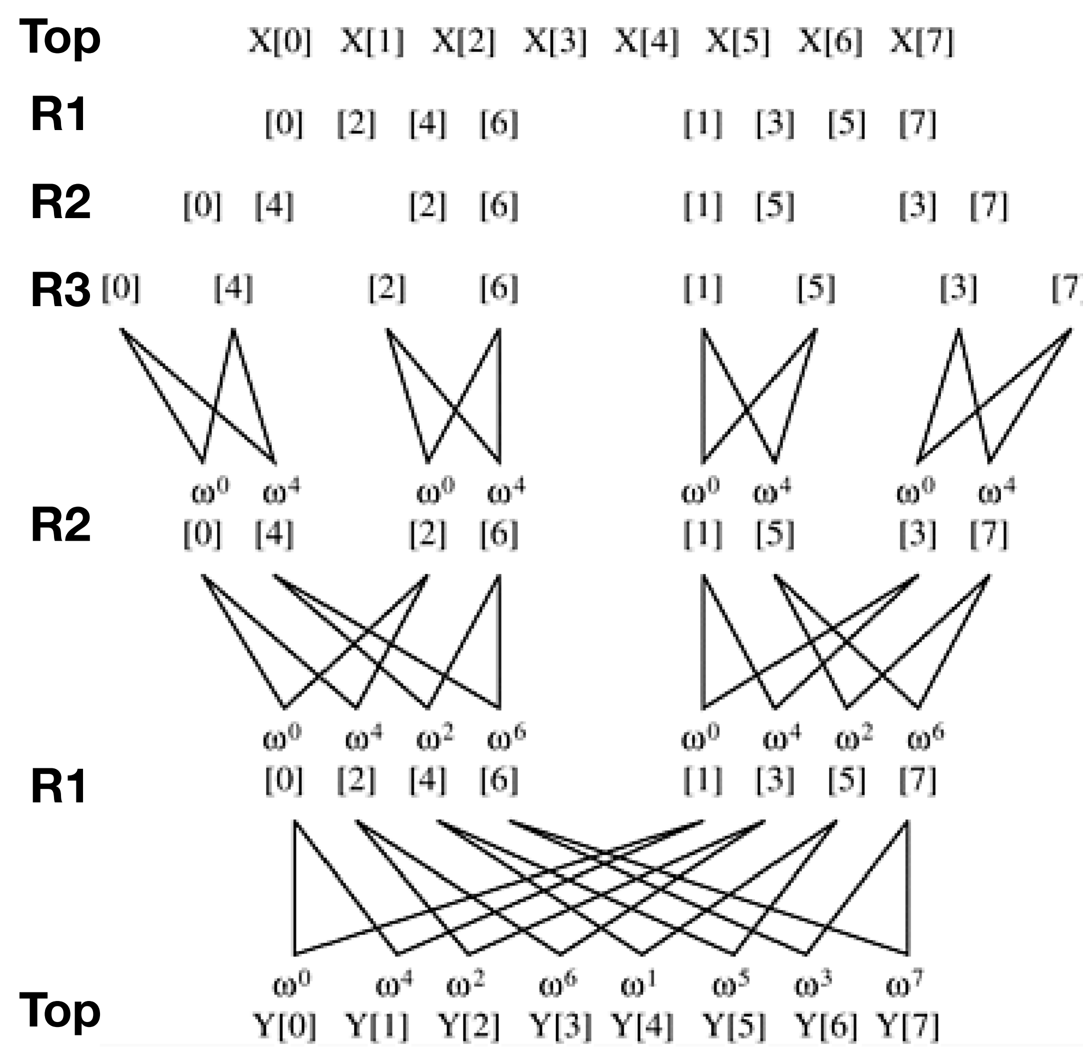
Runtime structure: Executor

$$X_k = \sum_{n=0}^{N-1} x_n e^{-\frac{2\pi i}{N} nk}$$

$$X_{N_2 k_1 + k_2} = \sum_{n_1=0}^{N_1-1} \sum_{n_2=0}^{N_2-1} x_{N_1 n_2 + n_1} e^{-\frac{2\pi i}{N_1 N_2} \cdot (N_1 n_2 + n_1) \cdot (N_2 k_1 + k_2)}$$

$$= \boxed{\sum_{n_1=0}^{N_1-1} \boxed{\left[e^{-\frac{2\pi i}{N} n_1 k_2} \right]} \boxed{\left(\sum_{n_2=0}^{N_2-1} x_{N_1 n_2 + n_1} e^{-\frac{2\pi i}{N_2} n_2 k_2} \right)} e^{-\frac{2\pi i}{N_1} n_1 k_1}}$$

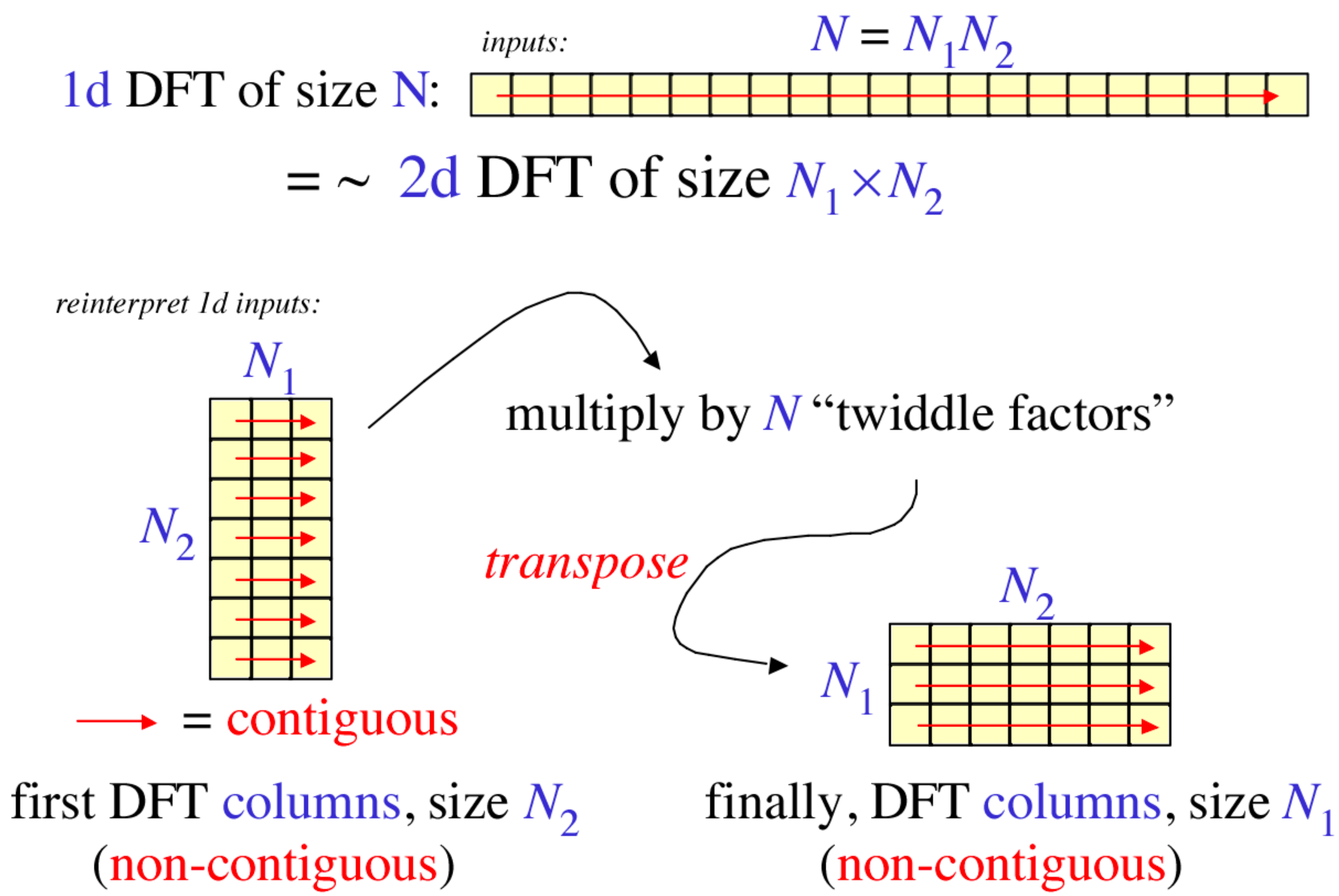
- Executor implements Cooley-Tukey FFT algorithm
  - Factors the size N of the transform into $N = N_1 N_2$
  - Recursively computes N_1 transforms of size N_2
  - Multiply the results by '*twiddle factors*'
  - Computes N_2 transforms of size N_1
- The algorithm mainly composed of two codelet variations (SIMD supported)
  - **Normal codelets**: Computes DFT of a fixed size and used as base case for recursion
  - **Twiddle codelets:** Like normal codelets, except they multiply their input by the twiddle factors



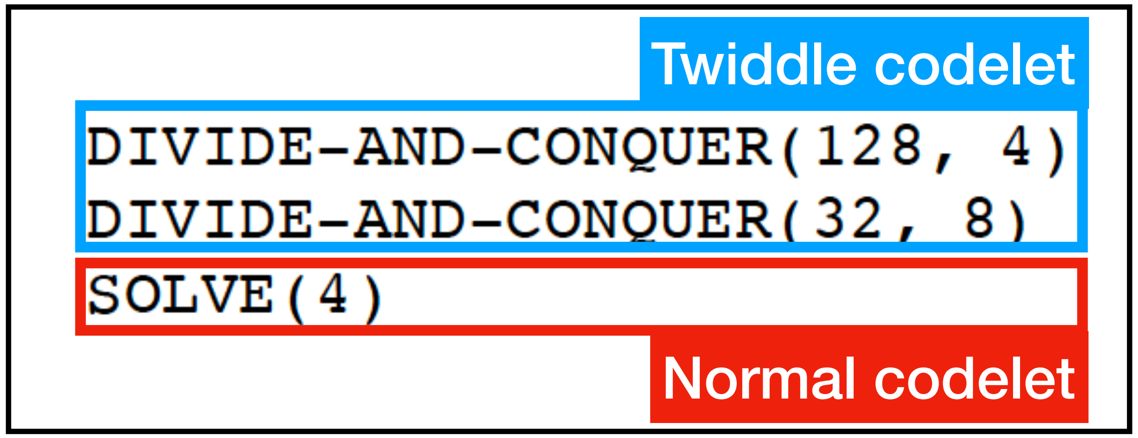
https://cw.fel.cvut.cz/wiki/_media/courses/b4m35pag/chap13_slides.pdf

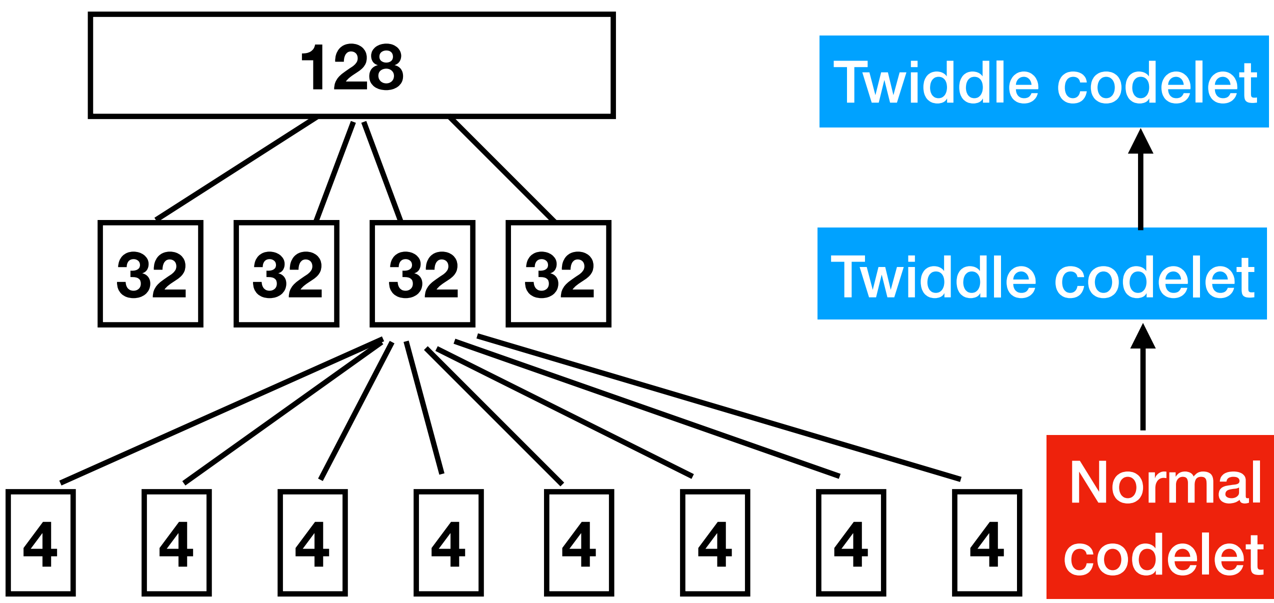
Runtime structure: Executor

- Codelet with SIMD support
- Need transpose for appropriate data layout
- Example SIMD scheme: DFT(A + *i*B) = DFT(A) + *i* DFT(B)



Runtime structure: Executor





- Input:
  1. Plan that specifies data structure of factorization and codelets to use
  2. Array to be transformed
- Example of a possible plan for a transform of length N=128
  - Computes 4 transforms of size 32 recursively then uses *twiddle codelet* of size 4 to combine results of the subproblems
  - Computes 8 transforms of size 4 solved directly using *normal codelet* and combined using size 8 twiddle codelet

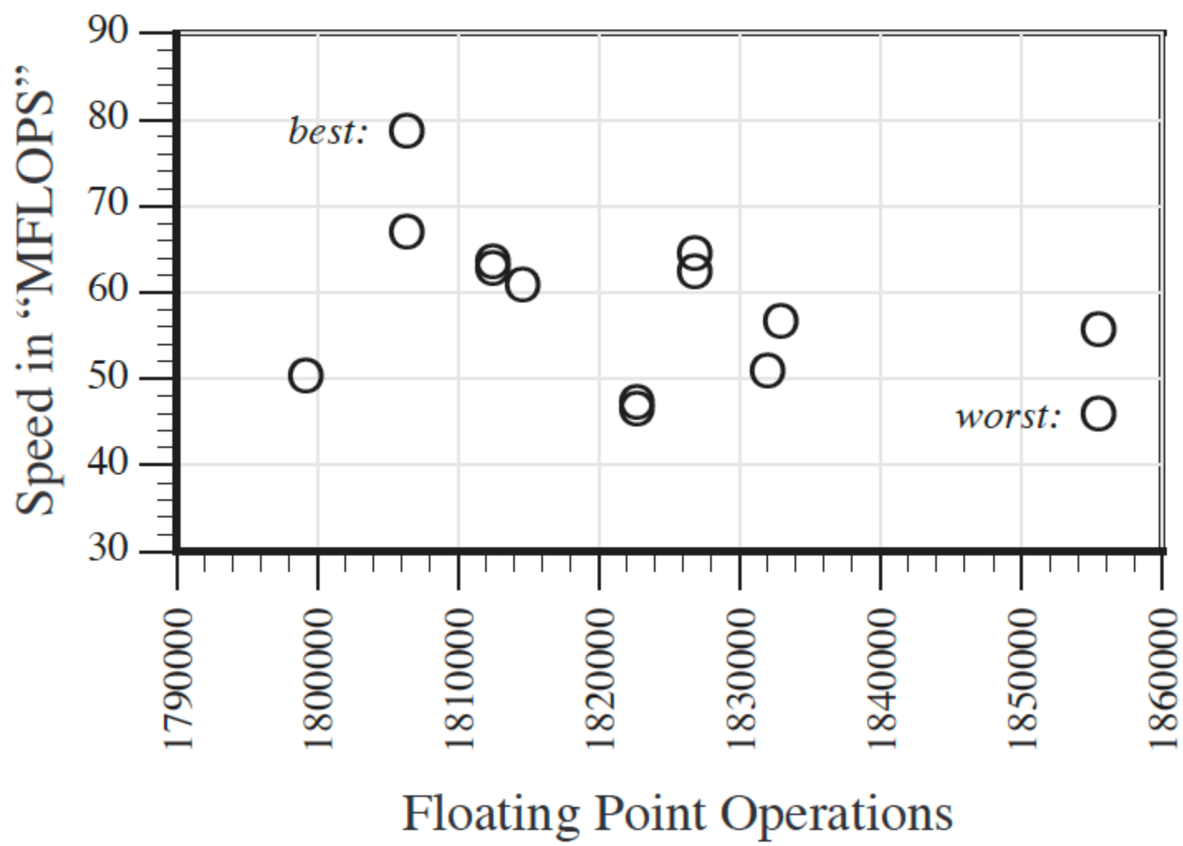
Runtime structure: Executor

- Implemented with explicit recursion instead of loop-based
  - Divide-and-conquer algorithms improve locality
  - Codelet performs significant amount of work - recursion overhead is negligible
  - Easier to code and allows codelet to perform well-defined task independent of the context

Runtime structure: Planner

- Strategy: Construct plan with combinations of codelets and measure execution time of different plans to select the best (ideally, all possible plan)
- Problem: Impractical due to combinatorial explosion of number of plans
- Solution: Use dynamic programming algorithm to reduce search space
  - Assume optimal sub-structure: *If an optimal plan for a size N is known, this plan is still optimal when size N is a subproblem of a larger transform*
  - In theory, the assumption is not true - cache states may differ
  - In practice, simplifying hypothesis yielded good results

Runtime structure: Planner



Speed of various plans as a function of number of flops required

- Fastest plan is not one that performs the fewest operations
- Total number of flops is not enough to predict execution time
- Optimal plan depends on processor, memory architecture and compiler
  - N = 1024 is factored into 8*8*16 on UltraSPARC and into 32*32 on Alpha

"MFLOPS" defined for a transform of size N as $(5N\log_2 N)/t$

Compile time: Codelet

- Codelet generator is written in Caml Light dialect of ML and is used during compile time
- Input: Size N
- Output: normal or twiddle codelet that performs Fourier transform of size N
- Operates on a subset of abstract syntax tree (AST) of the C language
- Codelet generation is broken down into three phases:
  - Generation: Creates a crude AST, contains *useless code*
  - Optimization/Simplification: Polish and apply local optimization on the crude AST
  - Scheduler: Topological sort of the AST to minimize register spills
  - Translation: Unparse the AST to produce desired C code

Codelet: Generation

- AST generator builds syntax tree recursively
  - Generator needs to decide which algorithm to use at each stage of recursion
    - split-radix - recursive split to N/2-N/4-N/4
    - prime factor - $N = N_1N_2$ where N_1N_2 prime numbers
    - Cooley-Tukey - $N = N_1N_2$
    - Rader's algorithm - Computes DFT of prime sizes
  - Minimize a certain cost function which depends on arithmetic complexity and memory traffic
    - Example: $cost = 4v + f$ (experimentally showed good results)
      - f is the number of floating-point operations
      - v is the number of stack variable

Codelet: Generation

$$\sum_{n_1=0}^{N_1-1} \left[e^{-\frac{2\pi i}{N} n_1 k_2} \right] \left(\sum_{n_2=0}^{N_2-1} x_{N_1 n_2 + n_1} e^{-\frac{2\pi i}{N_2} n_2 k_2} \right) e^{-\frac{2\pi i}{N_1} n_1 k_1}$$

```
let rec cooley_tukey n1 n2 input sign =
  let tmp1 j2 = fftgen n1
    (fun j1 -> input (j1 * n2 + j2)) sign in
  let tmp2 i1 j2 =
    exp n (sign * i1 * j2) @* tmp1 j2 i1 in
  let tmp3 i1 = fftgen n2 (tmp2 i1) sign
  in
   (fun i -> tmp3 (i mod n1) (i / n1))
```

Fragment of codelet generator that implements Cooley-Tukey

```
tmp1 = REAL(input[0]);
tmp5 = REAL(input[0]);
tmp6 = IMAG(input[0]);
tmp2 = IMAG(input[0]);
tmp3 = REAL(input[1]);
tmp7 = REAL(input[1]);
tmp8 = IMAG(input[1]);
tmp4 = IMAG(input[1]);
REAL(output[0]) = ((1 * tmp1) - (0 * tmp2))
      + ((1 * tmp3) - (0 * tmp4));
IMAG(output[0]) = ((1 * tmp2) + (0 * tmp1))
      + ((1 * tmp4) + (0 * tmp3));
REAL(output[1]) = ((1 * tmp5) - (0 * tmp6))
      + ((-1 * tmp7) - (0 * tmp8));
IMAG(output[1]) = ((1 * tmp6) + (0 * tmp5))
      + ((-1 * tmp8) + (0 * tmp7));
```

C translation of an AST for a complex DFT of size 2

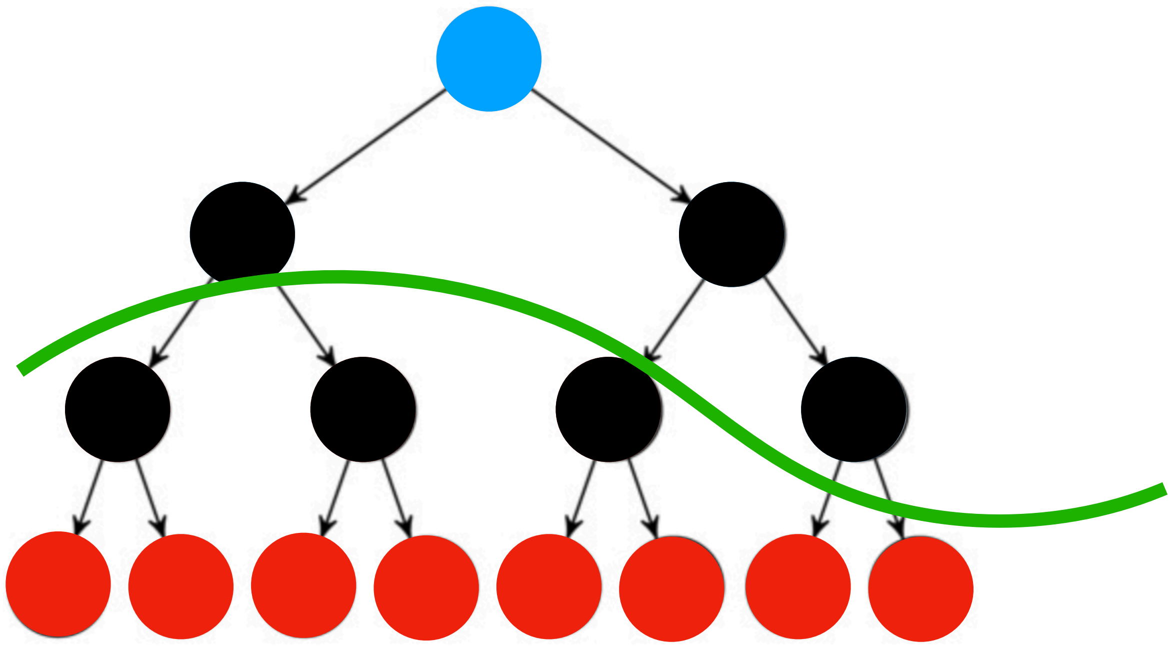
Codelet: Simplification

- Optimizer consists of a set of rules applied locally to each node in the AST to transform it into one that executes faster
- Codelet may contain many floating-point constant coefficients pair (i.e. a,-a) from trigonometric identities
  - Hack: Have a rule to make all constants positive and propagate the minus sign accordingly
  - Floating point constants are typically not part of the program code and are loaded from memory

```
let rec stimesM = function
  | (Uminus a, b) -> (* -a * b  ==> -(a * b) *)
     stimesM (a, b) >>= suminusM
  | (a, Uminus b) -> (* a * -b  ==> -(a * b) *)
     stimesM (a, b) >>= suminusM
  | (Num a, Num b) -> (* multiply two numbers *)
     snumM (Number.mul a b)
  | (Num a, Times (Num b, c)) ->
     snumM (Number.mul a b) >>= fun x ->
       stimesM (x, c)
  | (Num a, b) when Number.is_zero a ->
     snumM Number.zero   (* 0 * b  ==> 0 *)
  | (Num a, b) when Number.is_one a ->
     returnM b           (* 1 * b  ==> b *)
  | (Num a, b) when Number.is_mone a ->
     suminusM b          (* -1 * b  ==> -b *)
  | (a, (Num _ as b')) -> stimesM (b', a)
  | (a, b) -> returnM (Times (a, b))
```

Codelet: Scheduler

- Aim at maximizing register usage (Remember Hong and Kung?)
- However, codelet generator does not address instruction scheduling problem - pipelining
- Heuristic: Recursive partitioning



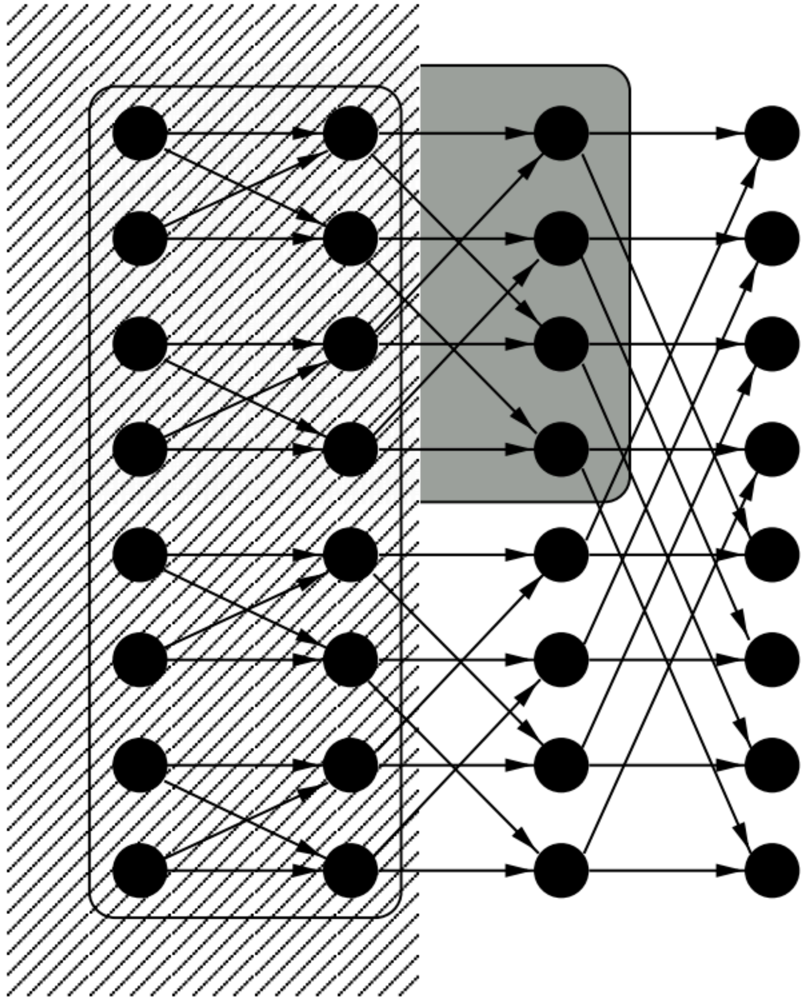
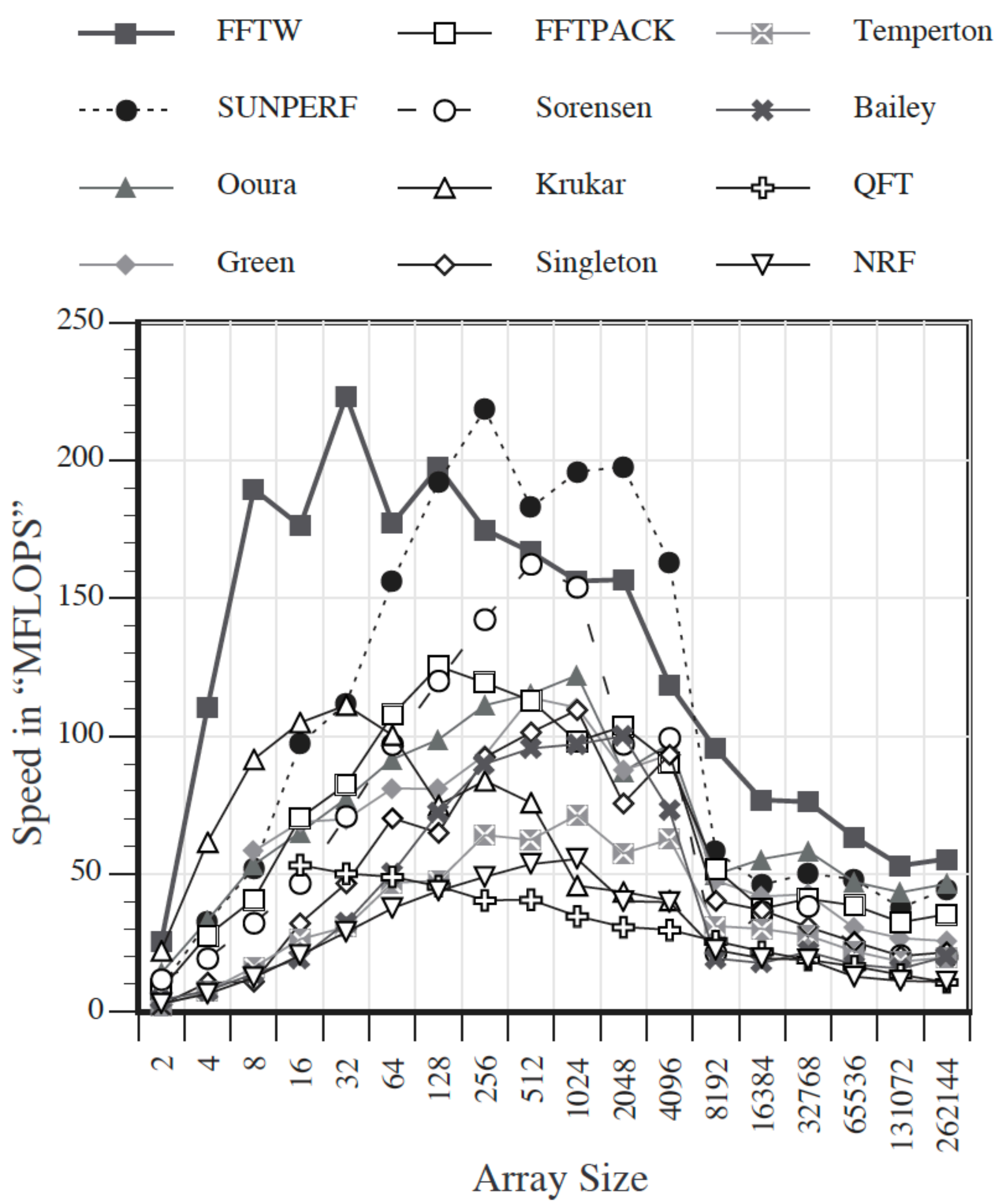


Illustration of a scheduling problem for FFT on 8 inputs

Compile time: Codelet

- Advantages of codelet generator:
  - Produce correct code automatically
  - Allows hacks such as propagation of the minus sign implemented with minimal code
  - Algorithm and coding style for best performance is not known *a priori*, generator helps produce and experiment code quickly

Performance results



- Compared FFTW with over 40 other complex FFT implementations on 7 platforms (not all is shown)
- Obtained similar numbers on other machines
  - On IBM RS/6000, comparing with IBM's ESSL library
    - N = 64, FFTW 55% faster
    - N=16384, FFTW 12% slower
    - N = 131072, FFTW 7% faster

Conclusion

- Manually optimizing software is impractical due to complexity of computer architecture
- FFTW provides a method to address such complexity by minimizing execution time instead of arithmetic complexity
    - Planner seeks for best execution plan
    - Codelet generator generates optimized code for transforms