# Experiences in coding high-performance numerical libraries

Matteo Frigo

Intel

April 18, 2010

# This lecture

- Talk about some things I learned while developing high-performance codes.
- Focus on numeric computations.
  - Linear algebra.
  - Stencil computations.
  - FFT.
- Caveat: experiments are somewhat IBM/PowerPC-centric.

# Coding cache oblivious algorithms

# Recursive matrix multiplication

**The usual description of the algorithm:**

Let $A$, $B$, and $C$ be $n \times n$ matrices. Want $C = AB$.

$$
\begin{aligned}
\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} &= \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \times \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} \\
&= \begin{bmatrix} A_{11}B_{11} & A_{11}B_{12} \\ A_{21}B_{11} & A_{21}B_{12} \end{bmatrix} + \begin{bmatrix} A_{12}B_{21} & A_{12}B_{22} \\ A_{22}B_{21} & A_{22}B_{22} \end{bmatrix}
\end{aligned}
$$

The size of the submatrices in $n/2 \times n/2$.

Theoretically a great algorithm: cache oblivious, easily parallelizable, etc.

# “Wrong” implementation

```
void matmul(n, A, B, C)
{
    if (n == 1) {
        C += A * B;
    } else {
        matmul(n/2, A11, B11, C11);
        matmul(n/2, A11, B12, C12);
        matmul(n/2, A21, B11, C21);
        matmul(n/2, A21, B12, C22);
        matmul(n/2, A12, B21, C11);
        matmul(n/2, A12, B22, C12);
        matmul(n/2, A22, B21, C21);
        matmul(n/2, A22, B22, C22);
    }
}
```

**Too limited:**

Works only for square matrices, and only for $n = 2^k$.

# Matrix multiplication viewed as traversal of a 3D "iteration space"

**Abstract matrix multiplication algorithm:**

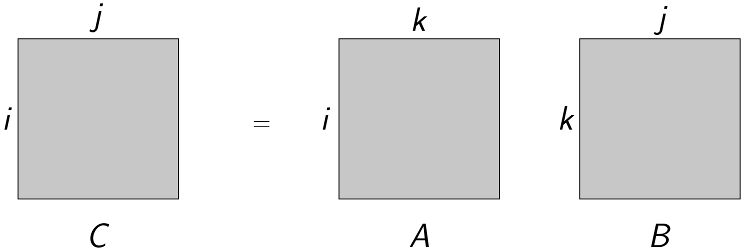
For all $(i, j, k)$ such that $i_0 \leq i < i_1$, $j_0 \leq j < j_1$, $k_0 \leq k < k_1$, in some unspecified order, do

$$\texttt{C}[i][j] \mathrel{+}= \texttt{A}[i][k] * \texttt{B}[k][j] \quad .$$

For square matrices, $i_0 = j_0 = k_0 = 0$ and $i_1 = j_1 = k_1 = n$.

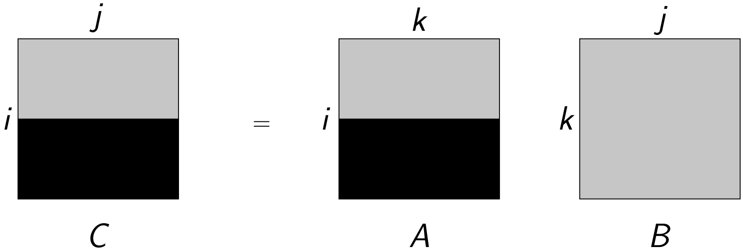
# Recursive traversal of the iteration space

Given arbitrary ranges of $i$, $j$, and $k$...



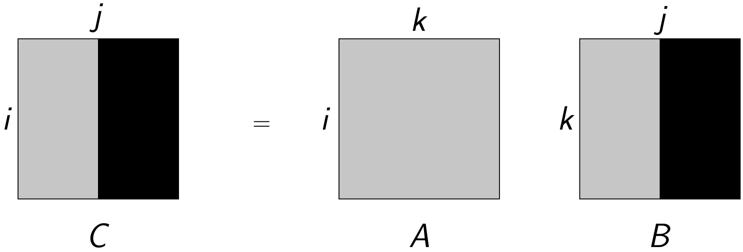
# Recursive traversal of the iteration space

If $i$ has the largest extent, cut $i$ and recur.



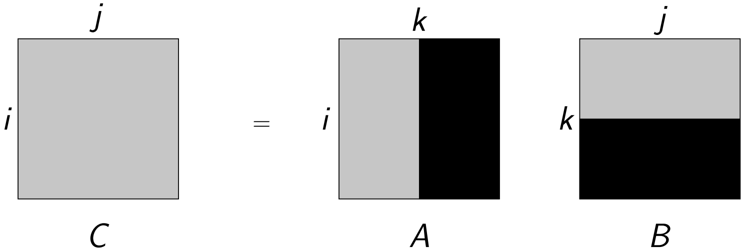
# Recursive traversal of the iteration space

If $j$ has the largest extent, cut $j$ and recur.



# Recursive traversal of the iteration space

If $k$ has the largest extent, cut $k$ and recur.



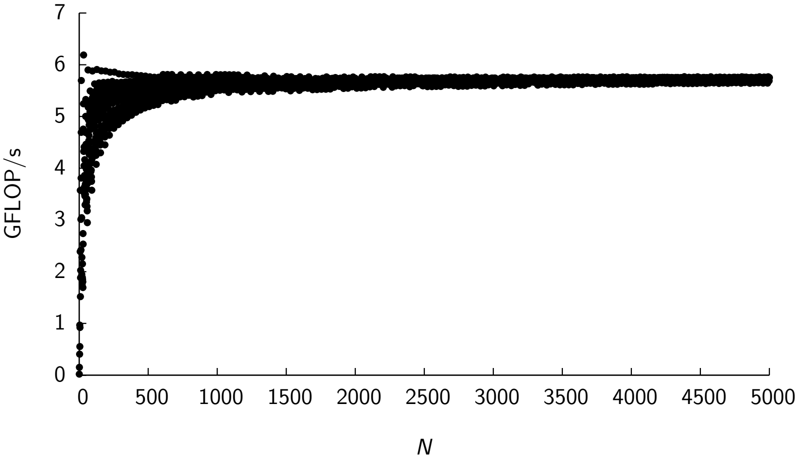

Always cut into **two** parts, not **eight**.

# Cache oblivious matrix multiplication code

```
void recur(int i0, int i1, int j0, int j1, int k0, int k1)
{
  int di = i1 - i0, dj = j1 - j0, dk = k1 - k0;
  const int CUTOFF = 8; /* "large enough" */
  if (di >= dj && di >= dk && di > CUTOFF) {
    int im = i0 + di/2;
    recur(i0, im, j0, j1, k0, k1);
    recur(im, i1, j0, j1, k0, k1);
  } else if (dj >= dk && dj > CUTOFF) {
    int jm = j0 + dj/2;
    recur(i0, i1, j0, jm, k0, k1);
    recur(i0, i1, jm, j1, k0, k1);
  } else if (dk > CUTOFF) {
    int km = k0 + dk/2;
    recur(i0, i1, j0, j1, k0, km);
    recur(i0, i1, j0, j1, km, k1);
  } else {
    base_case(i0, i1, j0, j1, k0, k1);
  }
}
```

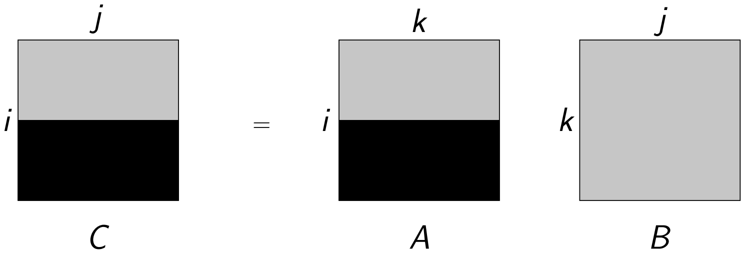
# Does the cache oblivious code work?

Performance of recursive matrix multiplication of $N \times N$ matrices, for all $1 \leq N < 5000$ on POWER5 (peak 6.6 Gflop/s).




(As good as any cache-aware code, *given the proper base case*.)

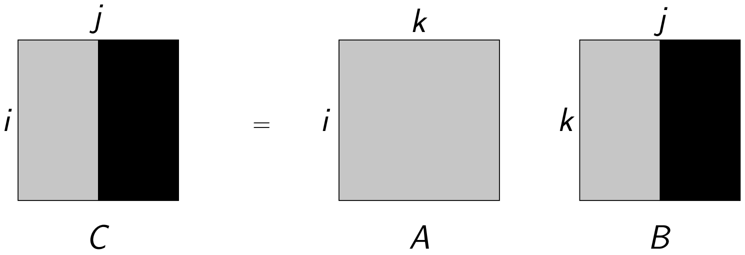
# Parallel matrix multiplication




*i*-**cut:**

- The two subproblems update **disjoint** locations of $C$.
- Can execute the two subproblems in parallel.

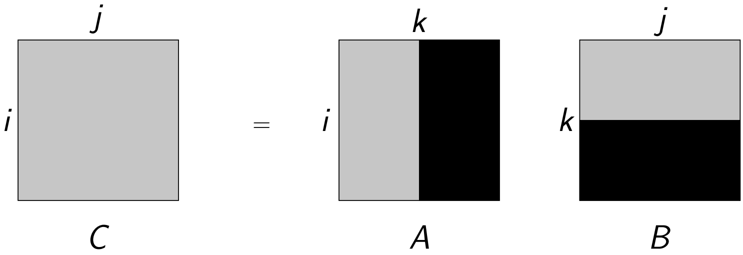
# Parallel matrix multiplication




**$j$-cut:**

- The two subproblems update **disjoint** locations of $C$.
- Can execute the two subproblems in parallel.

# Parallel matrix multiplication




**$k$-cut:**

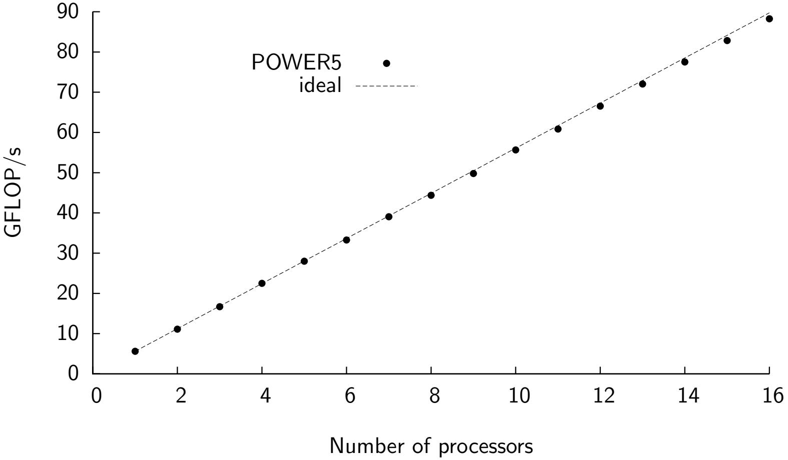
- The two subproblems update **overlapping** locations of $C$.
- Must execute the two subproblems sequentially.

# Parallel matrix multiplication code (Cilk++)

```
void recur(int i0, int i1, int j0, int j1, int k0, int k1)
{
  int di = i1 - i0, dj = j1 - j0, dk = k1 - k0;
  const int CUTOFF = 8; /* "large enough" */
  if (di >= dj && di >= dk && di > CUTOFF) {
    int im = i0 + di/2;
    cilk_spawn recur(i0, im, j0, j1, k0, k1);
    recur(im, i1, j0, j1, k0, k1);
  } else if (dj >= dk && dj > CUTOFF) {
    int jm = j0 + dj/2;
    cilk_spawn recur(i0, i1, j0, jm, k0, k1);
    recur(i0, i1, jm, j1, k0, k1);
  } else if (dk > CUTOFF) {
    int km = k0 + dk/2;
    recur(i0, i1, j0, j1, k0, km);
    recur(i0, i1, j0, j1, km, k1);
  } else {
    base_case(i0, i1, j0, j1, k0, k1);
  }
}
```

# Cilk parallel performance

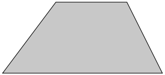
Performance for $N = 8000$ on up to 16 cores, using the MIT Cilk system.



# My experience

- The technique of recursive decomposition of the iteration space is widely applicable:
  - Linear algebra: matrix multiplication, LU decomposition, QR decomposition.
  - Matrix transposition.
  - Stencil computations.
  - All-pairs shortest path.
  - Dynamic programming: longest-common subsequence and other problems.
- Life is simpler if your recursive routine traverses a *trapezoid* rather than a *rectangle*.

Rectangle: 

Trapezoid: 

# Heat diffusion

### 1D heat diffusion equation:

$u(t,x)$: temperature at time $t$ at position $x$.

$$\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2} \,.$$

### Finite difference approximation:

$$\begin{aligned}
\frac{\partial u}{\partial x}(t,x) &\approx \frac{u(t,x+\Delta x/2) - u(t,x-\Delta x/2)}{\Delta x} \\
\frac{\partial^2 u}{\partial x^2}(t,x) &\approx \frac{(\partial u/\partial x)(t,x+\Delta x/2) - (\partial u/\partial x)(t,x-\Delta x/2)}{\Delta x} \\
&\approx \frac{u(t,x+\Delta x) - 2u(t,x) + u(t,x-\Delta x)}{(\Delta x)^2} \,.
\end{aligned}$$

# 3-point stencil

## Finite differences for the heat diffusion equation:

$$\frac{u(t+1,x_i)-u(t,x_i)}{\Delta t}=\frac{u(t,x_{i-1})-2u(t,x_i)+u(t,x_{i+1})}{(\Delta x)^2}\ .$$

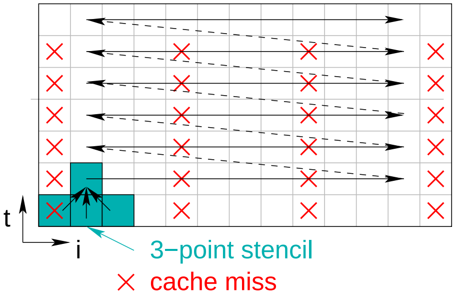
## Simple implementation:

```
for (t = 0; t < T; ++t) {                  /* time loop */
    u[(t+1)%2][0] = left_boundary();
    for (i = 1; i < N - 1; ++i)            /* space loop */
        u[(t+1)%2][i] =
           kernel(u[t%2][i-1], u[t%2][i], u[t%2][i+1]);
    u[(t+1)%2][N - 1] = right_boundary();
}

double kernel(u_{i-1}, u_i, u_{i+1})
{
    return u_i + Δt/(Δx)^2 * (u_{i-1} - 2*u_i + u_{i+1});
}
```

# 3-point stencil on a cache

```
for (t = 0; t < T; ++t) {                 /* time loop */
    u[(t+1)%2][0] = left_boundary();
    for (i = 1; i < N - 1; ++i)           /* space loop */
        u[(t+1)%2][i] =
           kernel(u[t%2][i-1], u[t%2][i], u[t%2][i+1]);
    u[(t+1)%2][N - 1] = right_boundary();
}
```




If array $u$ is larger than the cache, the number of misses is proportional to the number of accesses.

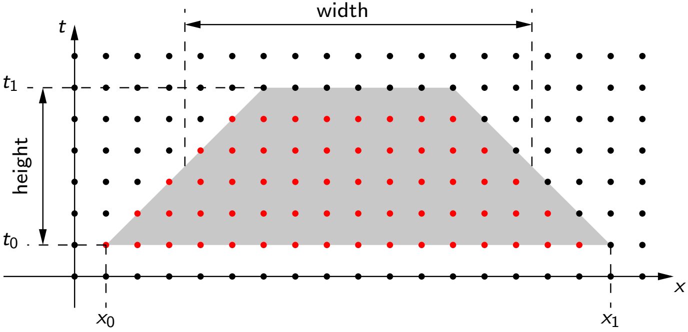
# Cache oblivious algorithm for 3-point stencil

Recursively traverse trapezoidal regions of spacetime points $(t, x)$ such that:

$$t_0 \leq t < t_1$$

$$x_0 + \dot{x}_0(t - t_0) \leq x < x_1 + \dot{x}_1(t - t_0)$$

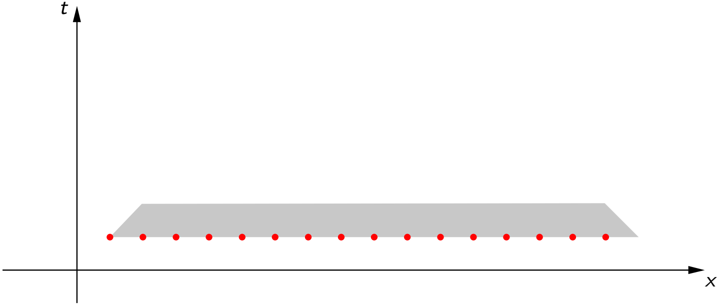
$$\dot{x}_i \in \{-1, 0, 1\}$$



# Base case

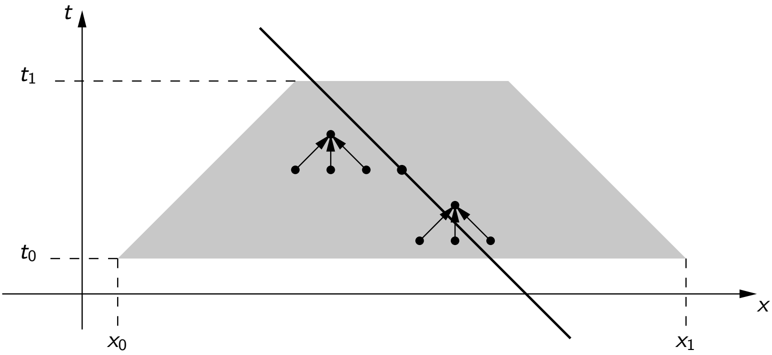
If height $= 1$, compute all spacetime points in the trapezoid.

Any order of computation is valid, because these points do not depend upon each other.



# Space cut

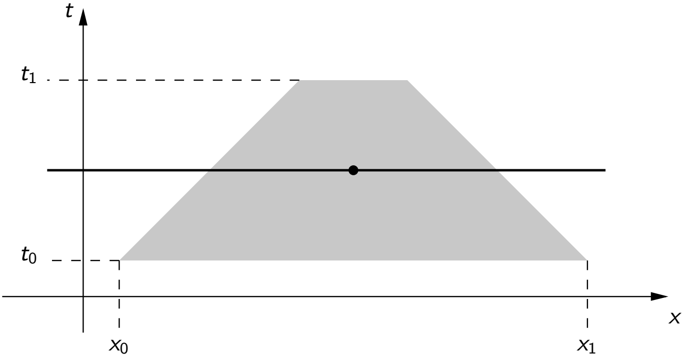
If width $\geq 2 \cdot$ height, cut the trapezoid with a line of slope $-1$ through the center.




Traverse first the trapezoid on the left, then the one on the right.

# Time cut

If width $< 2 \cdot$ height, cut the trapezoid with a horizontal line through the center.




Traverse the bottom trapezoid first, then the top one.

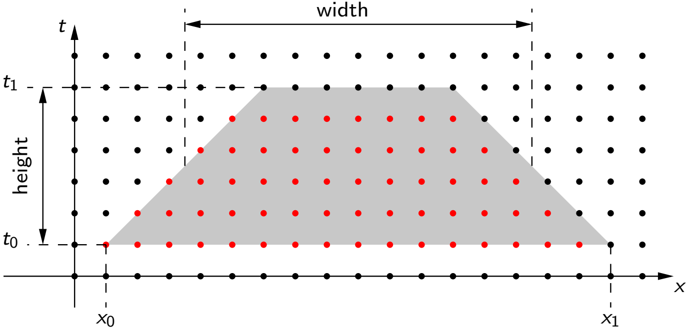
# C implementation

```
void trapezoid(int t0, int t1, int x0, int ẋ0, int x1, int ẋ1)
{
  int Δt = t1 - t0;
  if (Δt == 1) {
    int x;
    for (x = x0; x < x1; ++x)
      kernel(t0, x);
  } else if (Δt > 1) {
    if (2 * (x1 - x0) + (ẋ1 - ẋ0) * Δt >= 4 * Δt) {
      int xm = (2 * (x0 + x1) + (2 + ẋ0 + ẋ1) * Δt) / 4;
      trapezoid(t0, t1, x0, ẋ0, xm, -1);
      trapezoid(t0, t1, xm, -1, x1, ẋ1);
    } else {
      int s = Δt / 2;
      trapezoid(t0, t0 + s, x0, ẋ0, x1, ẋ1);
      trapezoid(t0 + s, t1, x0 + ẋ0 * s, ẋ0, x1 + ẋ1 * s, ẋ1);
    }
  }
}
```

# Cache complexity of the stencil algorithm

When width + height $= \Theta(Z)$:

- number of cache misses $= O(\text{width} + \text{height})$.
- number of points $= \Theta(\text{width} \cdot \text{height})$.
- Algorithm guarantees that height $= \Theta(\text{width})$.
- Thus, height $= \Theta(Z)$, width $= \Theta(Z)$.
- Thus, number of cache misses $= \Theta(\text{number of points}/Z)$.



# Demo

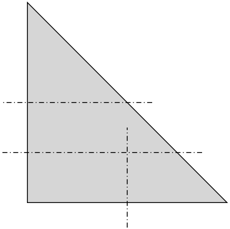
Simulation:

- $\Delta x = 95$.
- $\Delta t = 87$.
- $\dot{x}_0 = \dot{x}_1 = 0$.
- LRU cache.
- Line size = 4 points.
- Cache size = 4, 8, 16, or 32 cache lines.
- Cache miss latency = 10 cycles.

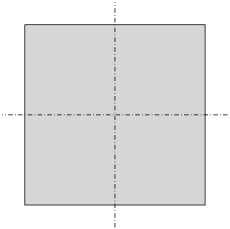
# Exercise

Program an in-place recursive matrix transposition routine in two ways:

1. Traversing the lower (or upper) triangle of the matrix.



2. Tiling a square matrix with squares.



# Choosing the input

# Lax-Wendroff code (3-point stencil)

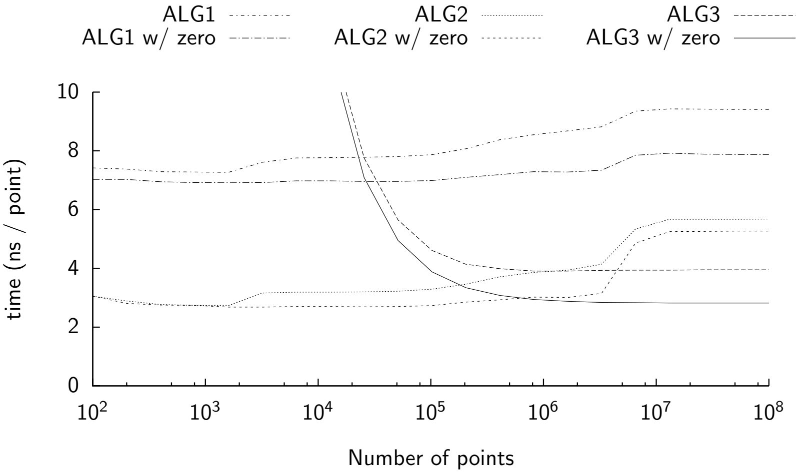
```
const double c = CONSTANT / 2.0;
const double c2 = CONSTANT * CONSTANT / 2.0;
double X[NUM_POINTS];

for (n=0; n<NSTEPS; n++) {
  double X_i_minus_1 = X[0];
  for (i=1; i<NUM_POINTS-1; i++) {
    double X_i = X[i];
    X[i] = X[i] - c * (X[i+1]-X_i_minus_1)
           + c2 * (X[i+1]-2.0*X[i]+X_i_minus_1);

    X_i_minus_1 = X_i;
  }
}
```

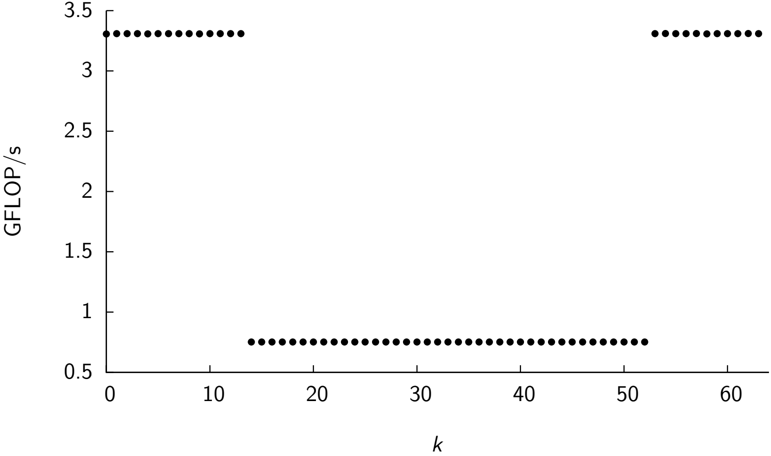
**Should run in** $O(\texttt{NSTEPS} * \texttt{NUM\_POINTS})$**, right?**

# Speed depends on the input values!




(On IBM POWER5.)

# Speed of $(1.0 + 2^{-k}) + (-1.0)$ (in double precision)




(On IBM POWER5. Also on PowerPC 970, a.k.a. Apple G5.)

# Floating-point numbers

## Floating-point representation

A floating-point number $x$ is represented as

$$x = m \cdot 2^e \ .$$

## Normalization condition

Floating-point numbers are (usually) normalized: $1/2 \leq m < 1$.

# Floating-point addition

## Example input

Let $m_1 = 10001$ (binary), $m_2 = -10000$. Assume same exponent $e = 0$.

## Step 1: Add

```
  10001  +
- 10000
---------
  00001
```

## Step 2: Normalize

- Find the most significant bit that is set and shift left.
- Before: $m = 00001$, $e = 0$.
- After: $m = 10000$, $e = -4$.

# Data-dependent FPU timing

## POWER5

- The POWER5 normalizer shifts by up to 16 positions in one cycle.
- Larger shifts take longer.

## x86 processors

- Huge slowdowns (100x) for denormalized numbers, infinities, NaNs, etc.

# My experience

- Hardware designers introduce irregularities for edge cases (by necessity).
- Nobody knows these irregularities.
  - Not even the “cycle accurate” simulator.
- To understand performance problems, you must know the details of your computer’s architecture.
- In practice: It is ok to set the input to zero for development purposes, but always verify with real data.

# Automatic generation of efficient code

# Automatic generation of computational kernels

*How do you optimize the base case of your matrix multiplication (or FFT, or stencil, or whatever)?*

**The hard way:**

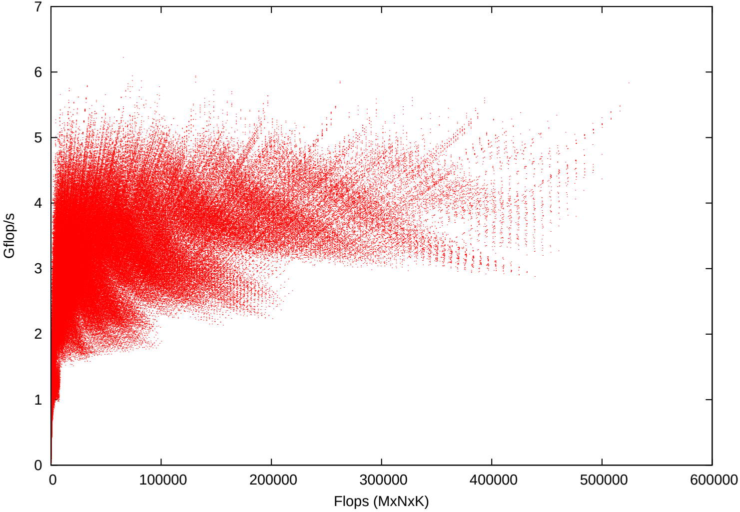
- Write code by hand and optimize it automatically using a general-purpose tool called a "compiler".
- If the compiler does not work, optimize your code by hand.

**The harder way:**

- Generate many "random" variants of a program and pick one that happens to run fast.
- There are way too many "random" programs. Better have some theory to restrict the search space, and hope that the theory is correct.
- Your compiler may not like the programs that you generate. You might have to write your own compiler as well.

# Good kernels are hard to find

Unrolled cache oblivious matrix multiplication kernels, $M \times K$ by $K \times N$ yielding $M \times N$, for all $M, N, K \in \{1, \ldots, 64\}$ on POWER5.



# Successful automatically-generated systems

## FFTW [Frigo and Johnson]

- Library for computing Fourier transforms.
- Generates hundreds of computational kernels (“codelets”) is a cache oblivious style.
- Finds a combination of codelets that happens to run fast on your machine.

## ATLAS [Whaley]

- Library for linear algebra.
- Generates many matrix-multiplication kernels trying to find a good one.
- Once found, it uses the kernel as much as possible.

# Kernel generators are conceptually simple

**Kernel ("does it"):**

```
for (i = 0; i < NI; ++i)
  for (j = 0; j < NJ; ++j)
    for (k = 0; k < NK; ++k)
      C[i][j] = A[i][k] * B[k][j];
```

**Kernel generator ("tells somebody else to do it"):**

```
for (i = 0; i < NI; ++i)
  for (j = 0; j < NJ; ++j)
    for (k = 0; k < NK; ++k)
      printf("C[%d][%d] = A[%d][%d] * B[%d][%d];\n",
                i, j, i, k, k, j);
```

# My experience with kernel generators

- Not too hard to write.
- I usually generate C (but also tried assembly).
- I have never been able to beat my own kernel generators, after appropriate exhaustive search. (Tried FFT, matrix multiplication, small convolutions, fast Walsh transform.)
- However, naive generators produce poor code.
- In particular, you *must* worry about register allocation.

# The register allocation problem

# An optimization that isn't

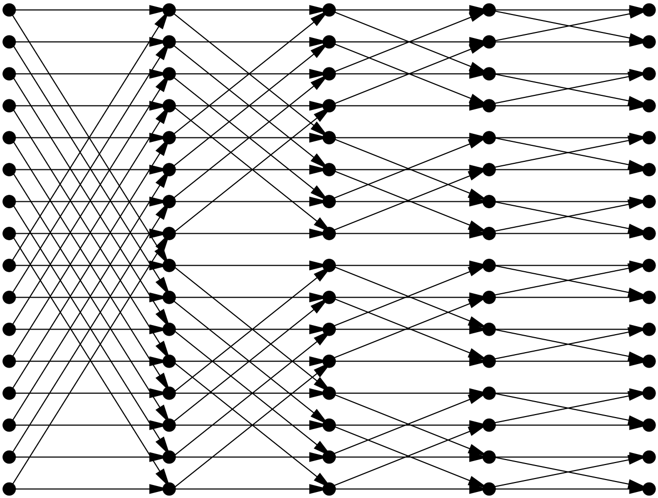
**32-point complex FFT in FFTW, PowerPC 7447 (year 2004)**

| add/sub | fma | load | store | code size | cycles |
|---|---|---|---|---|---|
| *C source:* | | | | | |
| 236 | 136 | 64 | 64 | ≈ 600 lines | |
| *Output of `gcc-3.4 -O2`:* | | | | | |
| 236 | 136 | 484 | 285 | 5620 bytes | ≈ 1550 |
| *Output of `gcc-3.4 -O2 -fno-schedule-insns`:* | | | | | |
| 236 | 136 | 134 | 125 | 2868 bytes | ≈ 640 |

- Disabling the gcc `schedule-insns` "optimization" improves performance by 2.5×.

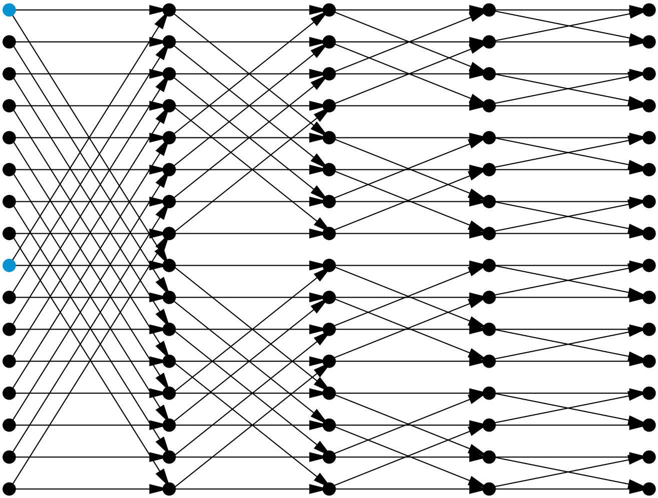
# What `gcc -fschedule-insns` does

CPU with 4 registers computes this graph:



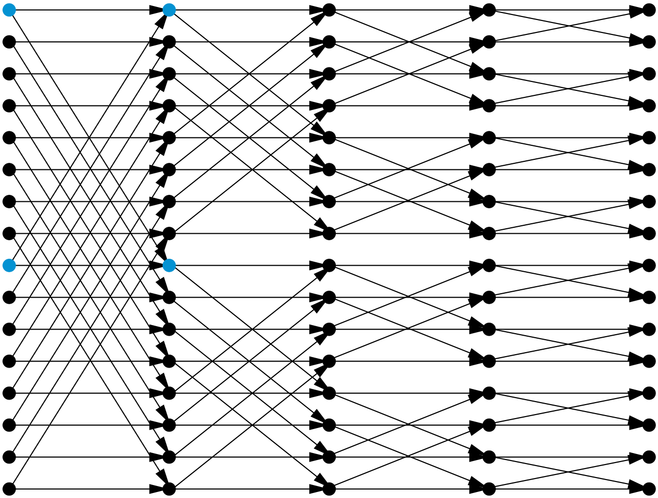
# What `gcc -fschedule-insns` does

Load two inputs into registers.



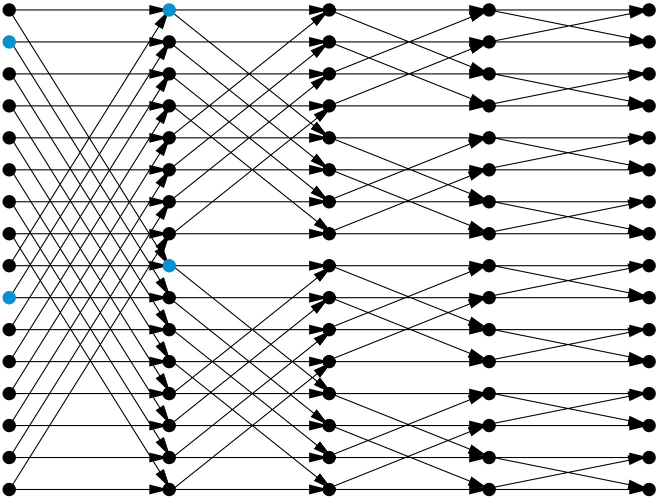
# What `gcc -fschedule-insns` does

Compute two nodes, in registers.



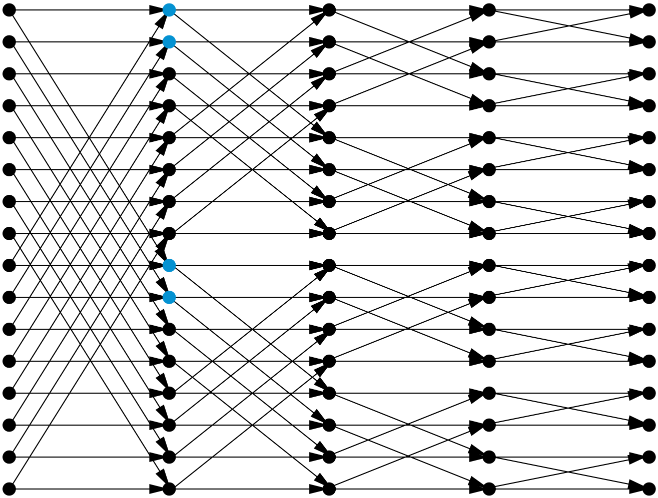
# What `gcc -fschedule-insns` does

Load two more inputs into registers.



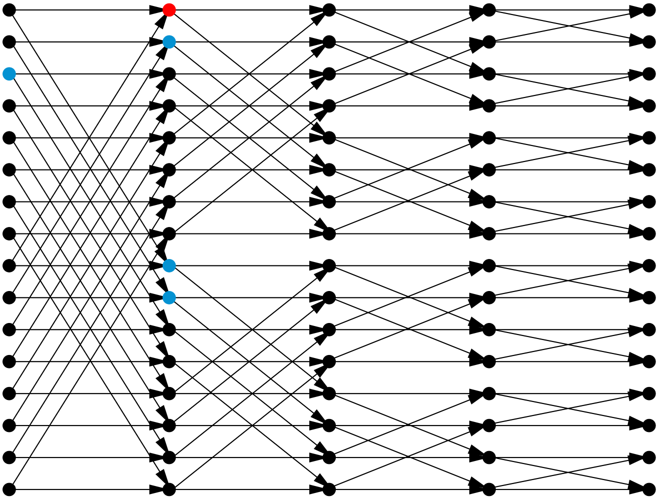
# What `gcc -fschedule-insns` does

Compute two more nodes, in registers.



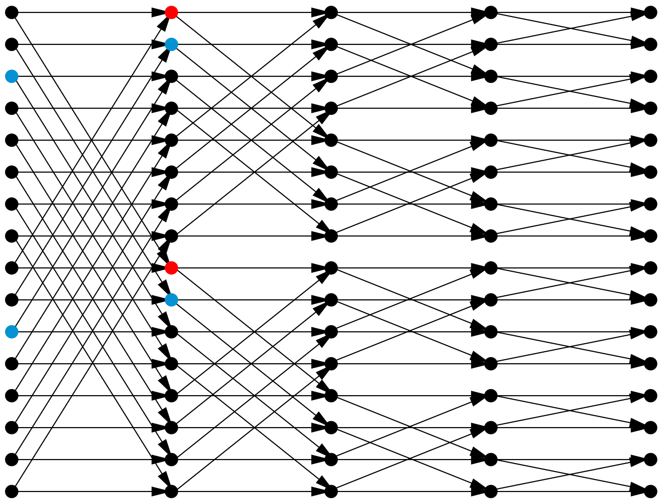
# What `gcc -fschedule-insns` does

Load another input. Must “spill” one register.



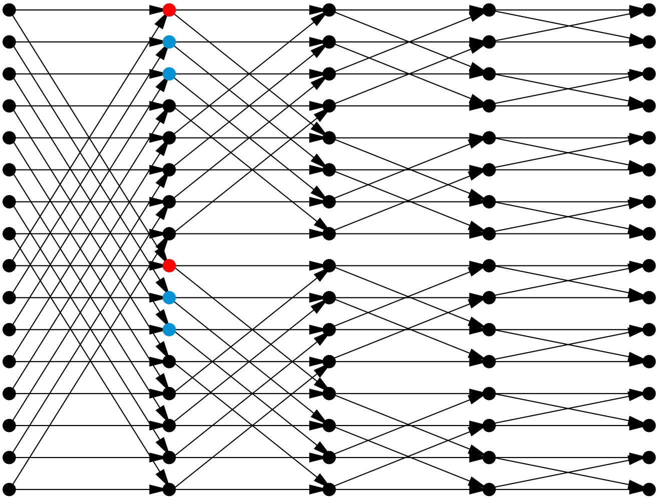
# What `gcc -fschedule-insns` does

Load another input. Must spill another register.



# What `gcc -fschedule-insns` does

Compute two more nodes, in registers.

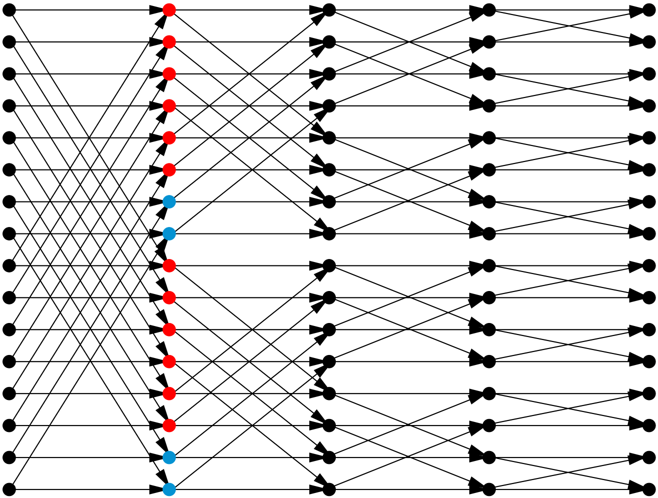


# What `gcc -fschedule-insns` does

*keep going for a while...*

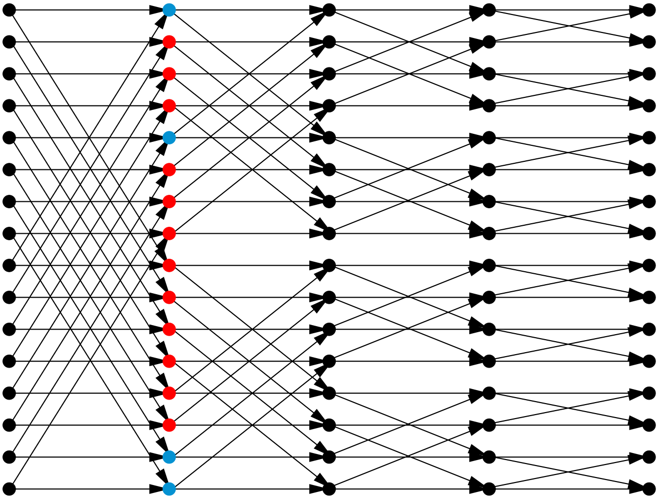
# What `gcc -fschedule-insns` does

4 values in registers, 12 values spilled.



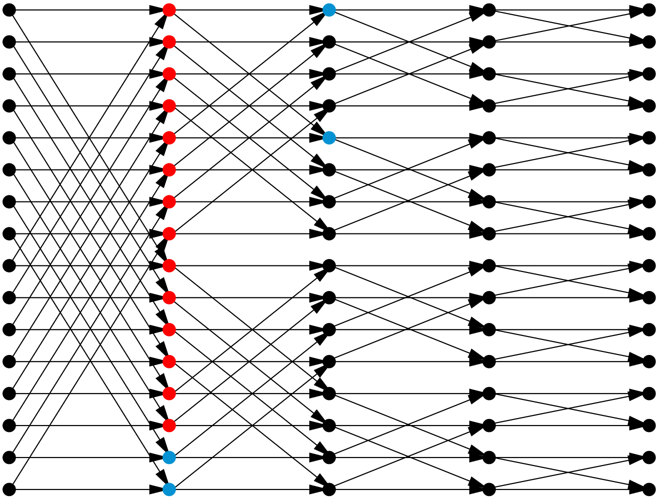
# What `gcc -fschedule-insns` does

Load one spilled value. Must spill one register.



# What `gcc -fschedule-insns` does

Compute two nodes, in registers. Etc.



# Why the `gcc` strategy cannot work

**Theorem**

*If*

- *you compute the FFT level by level, like gcc; and*
- *$n$ = number of inputs $\gg$ number of registers*

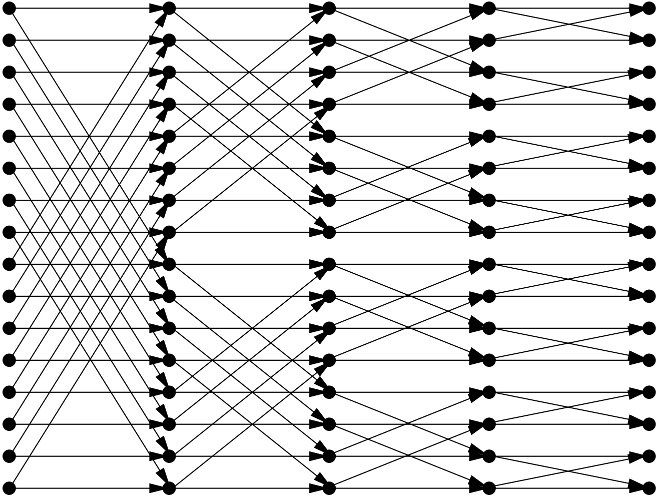
*then*

- *you must pay $\Theta(n \log n)$ register spills irrespective of how you allocate registers.*

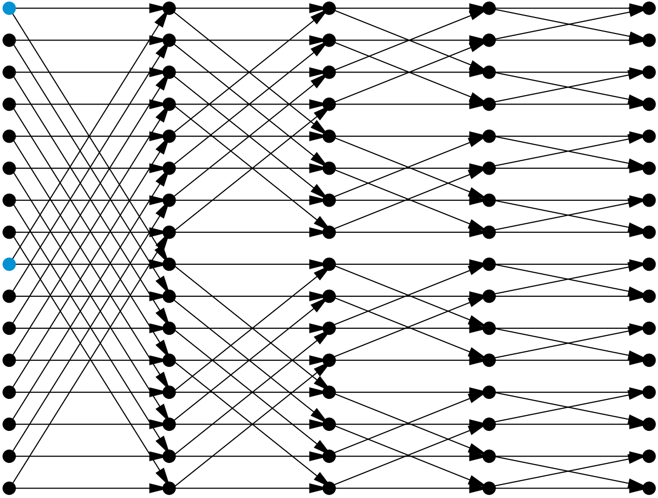
**Corollary**

*Because the FFT requires $\Theta(n \log n)$ operations, you pay $\Theta(1)$ spills per useful operation, no matter how many registers the machine has.*

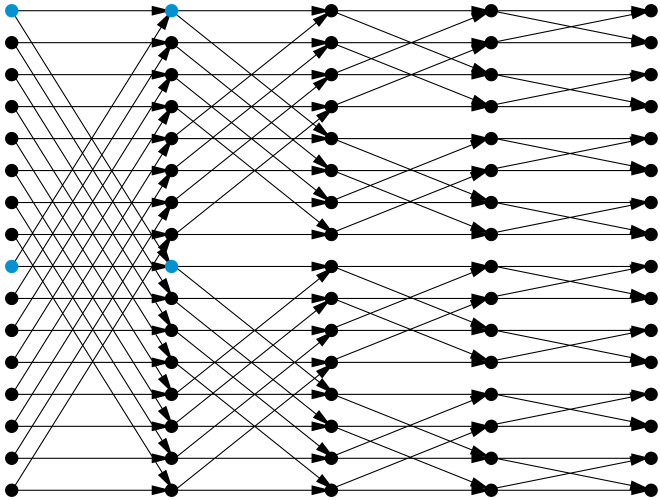
# Better strategy: blocking



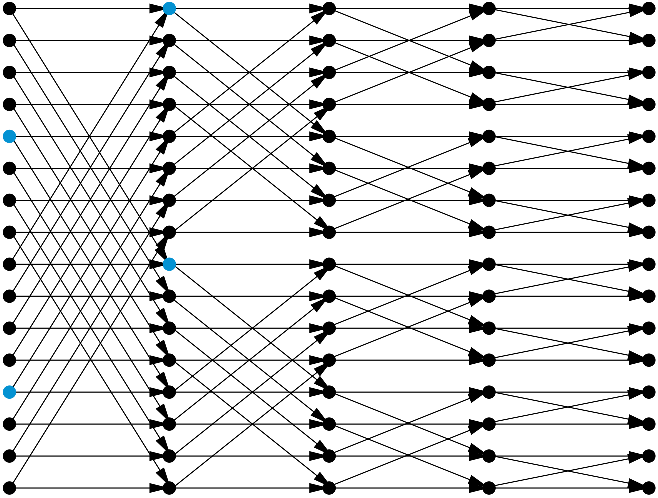
# Better strategy: blocking



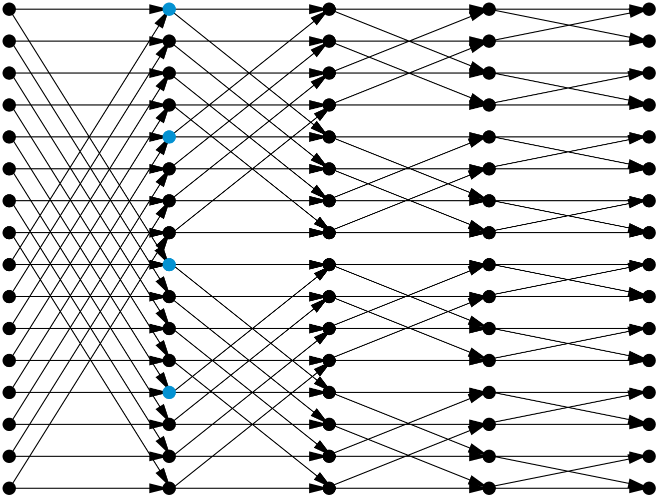
# Better strategy: blocking



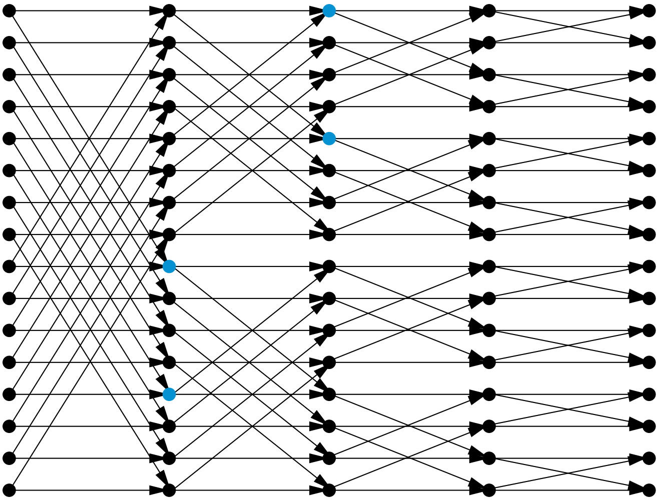
# Better strategy: blocking



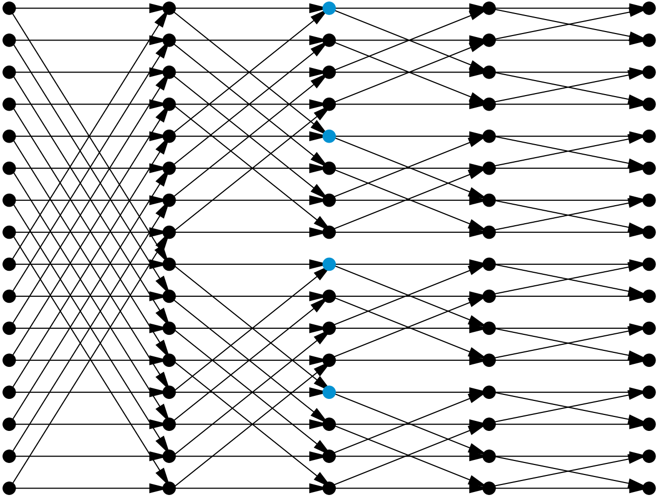
# Better strategy: blocking



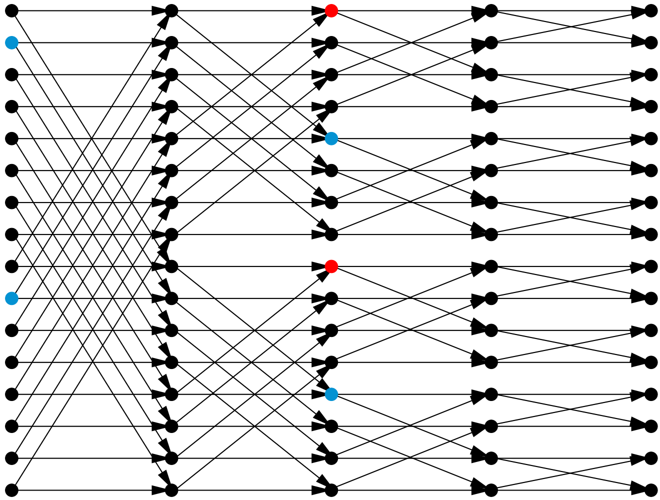
# Better strategy: blocking



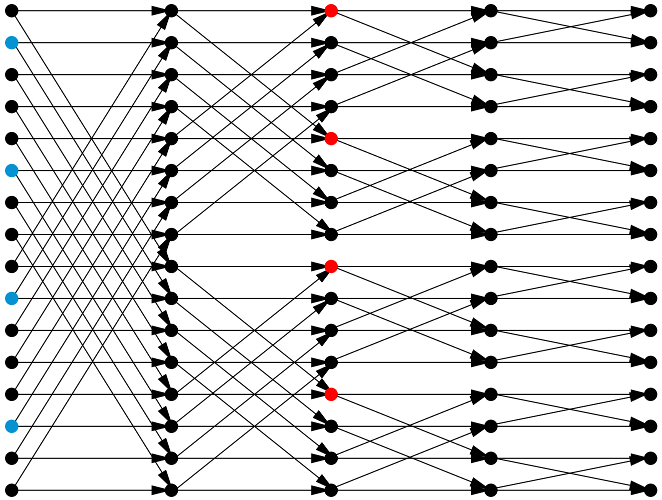
# Better strategy: blocking



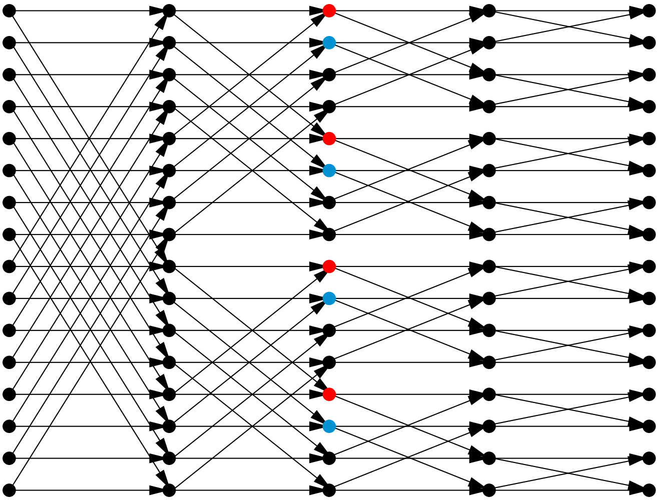
# Better strategy: blocking



# Better strategy: blocking



# Better strategy: blocking



# Analysis of the blocking schedule

**Theorem (Upper bound)**

*With $R$ registers,*

- *a schedule exists such that*
- *a register allocation exists such that*
- *the execution incurs $O(n \log n / \log R)$ register spills.*

**Theorem (Lower bound, Hong and Kung '81)**

*Any execution of the FFT graph with $R$ registers incurs $\Omega(n \log n / \log R)$ register spills.*

**Corollary**

*The blocking schedule is asymptotically optimal.*

# Complexity of register allocation

**Theorem (Motwani et al., 1995)**

*Given a dag, find both a schedule of the dag and a register assignment that minimizes the number of register spills:* **NP-hard**.

**Theorem (Belady 1966)**

*Given a dag and a schedule of the dag, find register assignment that minimizes the number of register spills:* $\approx$ **linear time**.

**Corollary**

- *It is unreasonable to expect a compiler to take an arbitrary dag and produce good code.*
- *However, if you schedule the dag yourself, a decent compiler should produce good code.*

# How do you find a good schedule?

**Key insight:**

- Registers are a cache managed by the compiler.
- Thus, techniques for optimizing the memory hierarchy (including cache oblivious algorithms) yield good schedules.
- If your kernel is straight-line, the cache is *ideal*: fully associative and with optimal replacement policy.

**In practice:**

- If you are lucky, the compiler might respect your schedule and approximate an ideal cache.
- Otherwise, you can implement Belady's algorithm yourself. (Easy to do.)

# Belady's register allocation

**Belady's policy:**

When you must evict a register, evict one used furthest in the future.

**Example (matrix multiplication, 4 registers):**

| | $r_0$ | $r_1$ | $r_2$ | $r_3$ |
|---|---|---|---|---|
| $c_{00} \leftarrow c_{00} + a_{00} \cdot b_{00}$ | $a_{00}$ | $b_{00}$ | $c_{00}$ | |
| $c_{00} \leftarrow c_{00} + a_{01} \cdot b_{10}$ | $a_{00}$ | $b_{10}$ | $c_{00}$ | $a_{01}$ |
| $c_{01} \leftarrow c_{01} + a_{00} \cdot b_{01}$ | $a_{00}$ | $b_{01}$ | $c_{01}$ | $a_{01}$ |
| $c_{01} \leftarrow c_{01} + a_{01} \cdot b_{11}$ | $b_{11}$ | $b_{01}$ | $c_{01}$ | $a_{01}$ |
| $c_{10} \leftarrow c_{10} + a_{10} \cdot b_{00}$ | $b_{00}$ | $b_{01}$ | $c_{10}$ | $a_{10}$ |
| $c_{10} \leftarrow c_{10} + a_{11} \cdot b_{10}$ | $b_{10}$ | $b_{01}$ | $c_{10}$ | $a_{11}$ |
| $c_{11} \leftarrow c_{11} + a_{10} \cdot b_{01}$ | $a_{10}$ | $b_{01}$ | $c_{11}$ | $a_{11}$ |
| $c_{11} \leftarrow c_{11} + a_{11} \cdot b_{11}$ | $b_{11}$ | $b_{01}$ | $c_{11}$ | $a_{11}$ |

(blue = load, red = spill.)

# Generated code with register allocation

```
r0 = a00
r1 = b00
r2 = c00
r2 = r2 + r0 * r1
r1 = b10
r3 = a01
r2 = r2 + r3 * r1
c00 = r2
r2 = c01
r1 = b01
r2 = r2 + r0 * r1
r0 = b11
r2 = r2 + r3 * r0
c01 = r2
r2 = c10
r0 = b00
r3 = a10
r2 = r2 + r3 * r0
r0 = b10
r3 = a11
r2 = r2 + r3 * r0
c10 = r2
r2 = c11
r0 = a10
r2 = r2 + r0 * r1
r0 = b11
r2 = r2 + r3 * r0
c11 = r2
```

# Summary

- Forget the algebra, think iteration space.
- Know what you are trying to measure.
- When in doubt, use brute force.
- Register allocation is just another exercise in caching.