How to Write Fast Numerical Code

Spring 2011

Lecture 5

Instructor: Markus Püschel

TA: Georg Ofenbeck

ETH

Eidgenössische Technische Hochschule Zürich

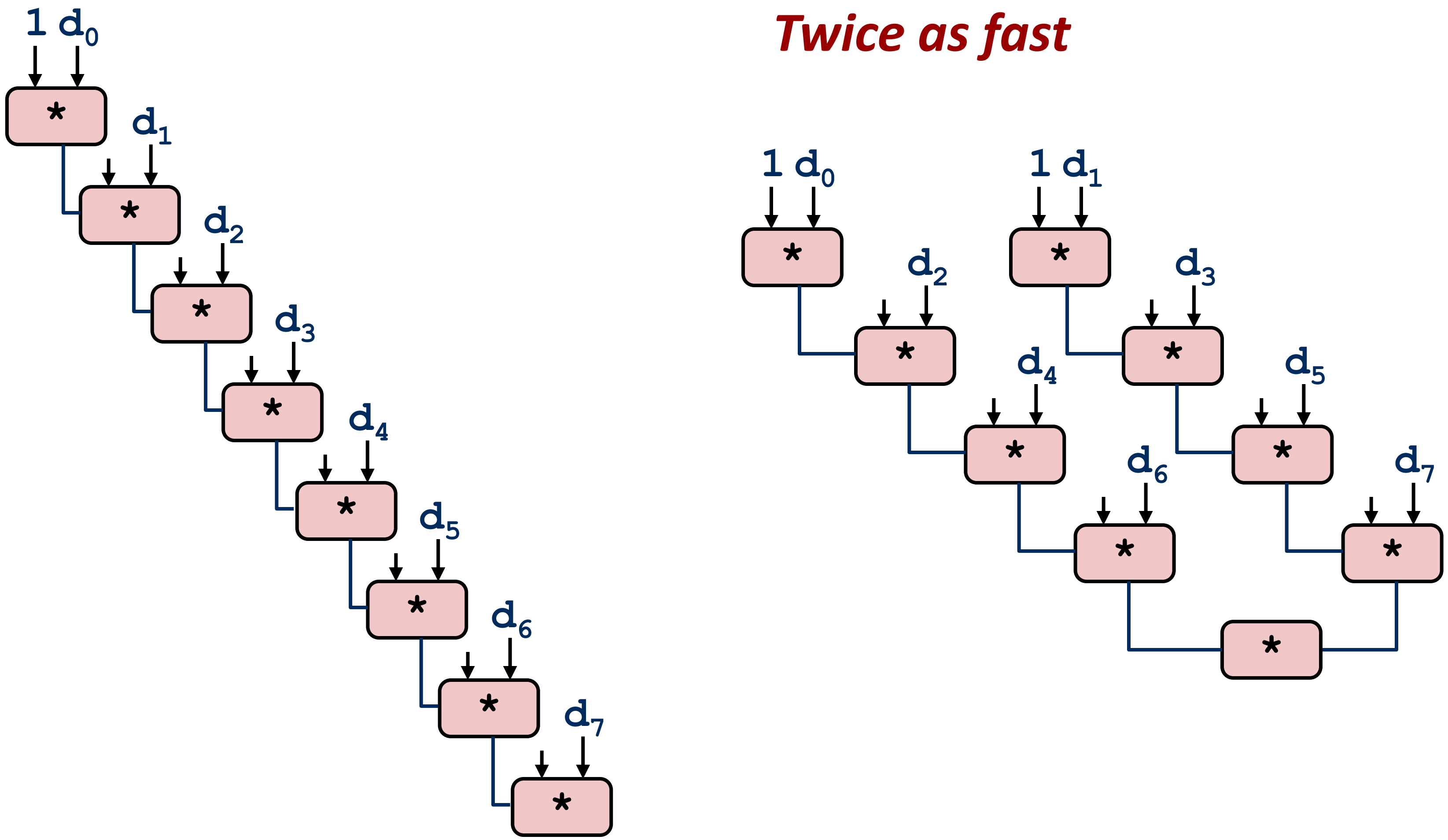
Swiss Federal Institute of Technology Zurich

Organizational

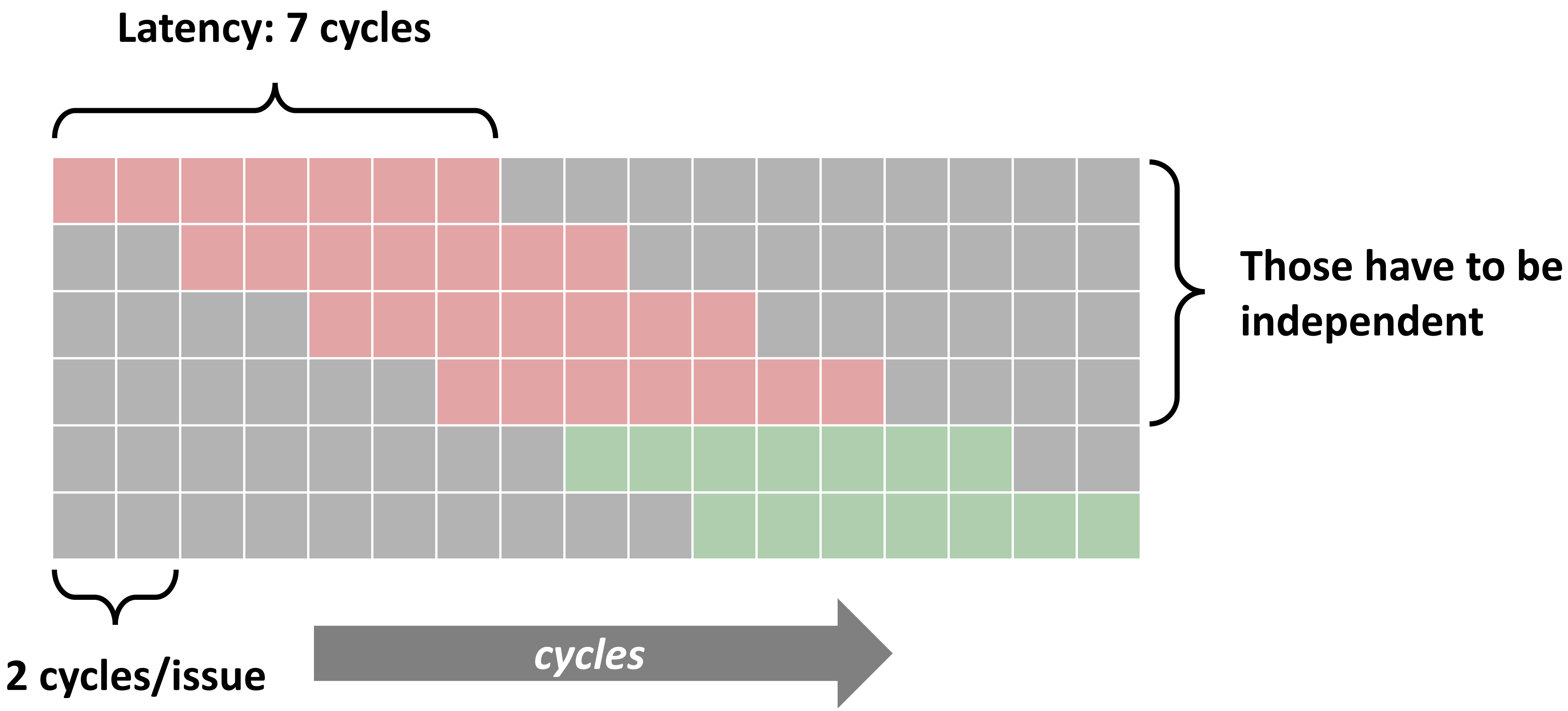
- *Class Monday 14.3. → Friday 18.3*
- **Office hours:**
  - Markus: Tues 14–15:00
  - Georg: Wed 14–15:00
- **Research projects**
  - 11 groups, 23 people
  - I need to approve the projects

Last Time: ILP

- **Latency/throughput (Pentium 4 fp mult: 7/2)**



Last Time: Why ILP?



Based on this insight: **K = #accumulators = ceil(latency/cycles per issue)**

Organization

- Instruction level parallelism (ILP): an example
- **Optimizing compilers and optimization blockers**
  - Overview
  - Removing unnecessary procedure calls
  - Code motion
  - Strength reduction
  - Sharing of common subexpressions
  - ***Optimization blocker: Procedure calls***
  - Optimization blocker: Memory aliasing
  - Summary

Compiler is likely to do that

Optimization Blocker #1: Procedure Calls

- Procedure to convert string to lower case

```
void lower(char *s)
{
  int i;
  for (i = 0; i < strlen(s); i++)
    if (s[i] >= 'A' && s[i] <= 'Z')
      s[i] -= ('A' - 'a');
}
```

$O(n^2)$ instead of $O(n)$

```
/* My version of strlen */
size_t strlen(const char *s)
{
    size_t length = 0;
    while (*s != '\0') {
        s++;
        length++;
    }
    return length;
}
```

$O(n)$

Improving Performance

```
void lower(char *s)
{
  int i;
  for (i = 0; i < strlen(s); i++)
    if (s[i] >= 'A' && s[i] <= 'Z')
      s[i] -= ('A' - 'a');
}
```

```
void lower(char *s)
{
  int i;
  int len = strlen(s);
  for (i = 0; i < len; i++)
    if (s[i] >= 'A' && s[i] <= 'Z')
      s[i] -= ('A' - 'a');
}
```

- **Move call to `strlen` outside of loop**
- **Since result does not change from one iteration to another**
- **Form of code motion/precomputation**

Optimization Blocker: Procedure Calls

- **Why couldn't compiler move `strlen` out of inner loop?**
  - Procedure may have side effects

- ***Compiler usually treats procedure call as a black box that cannot be analyzed***
  - Consequence: conservative in optimizations

- **In this case the compiler may actually do if `strlen` is recognized as built-in function**

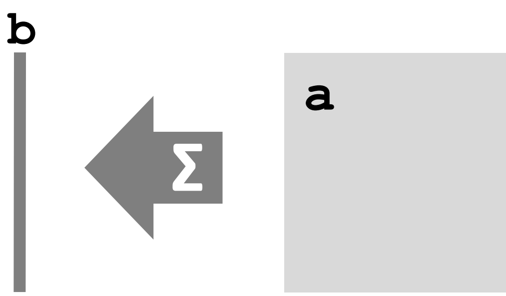
Organization

- Instruction level parallelism (ILP): an example
- **Optimizing compilers and optimization blockers**
  - Overview
  - Removing unnecessary procedure calls
  - Code motion
  - Strength reduction
  - Sharing of common subexpressions
  - Optimization blocker: Procedure calls
  - *Optimization blocker: Memory aliasing*
  - Summary

Compiler is likely to do that

Optimization Blocker: Memory Aliasing

```
/* Sums rows of n x n matrix a
   and stores in vector b  */
void sum_rows1(double *a, double *b, long n) {
    long i, j;
    for (i = 0; i < n; i++) {
        b[i] = 0;
        for (j = 0; j < n; j++)
            b[i] += a[i*n + j];
    }
}
```



- Code updates `b[i]` (= memory access) on every iteration
- Does compiler optimize this away? *No!*

Reason: Possible Memory Aliasing

- If memory is accessed, compiler assumes the possibility of side effects
- Example:

```
/* Sums rows of n x n matrix a
   and stores in vector b  */
void sum_rows1(double *a, double *b, long n) {
    long i, j;
    for (i = 0; i < n; i++) {
        b[i] = 0;
        for (j = 0; j < n; j++)
            b[i] += a[i*n + j];
    }
}
```

```
double A[9] =
  { 0,   1,   2,
    4,   8,  16},
   32,  64, 128};

double B[3] = A+3;

sum_rows1(A, B, 3);
```

Value of B:

```
init:   [4,  8,  16]

i = 0:  [3,  8,  16]

i = 1:  [3, 22,  16]

i = 2:  [3, 22, 224]
```

Removing Aliasing

```
/* Sums rows of n x n matrix a
   and stores in vector b  */
void sum_rows2(double *a, double *b, long n) {
    long i, j;
    for (i = 0; i < n; i++) {
        double val = 0;
        for (j = 0; j < n; j++)
            val += a[i*n + j];
         b[i] = val;
    }
}
```

- **Scalar replacement:**
  - Copy array elements ***that are reused*** into temporary variables
  - Perform computation on those variables
  - Enables register allocation and instruction scheduling
  - Assumes no memory aliasing (otherwise possibly incorrect)

Optimization Blocker: Memory Aliasing

- **Memory aliasing:**
  Two different memory references write to the same location

- **Easy to have happen in C**
  - Since allowed to do address arithmetic
  - Direct access to storage structures

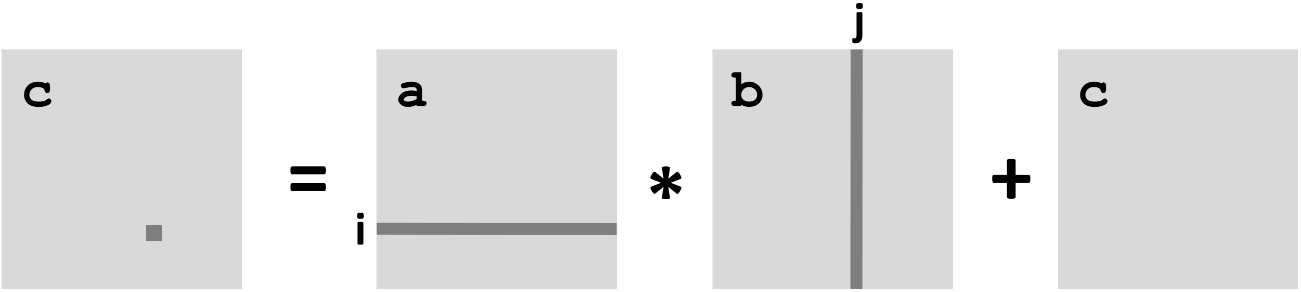
- **Hard to analyze = compiler cannot figure it out**
  - Hence is conservative

- **Solution: Scalar replacement in innermost loop**
  - Copy memory variables ***that are reused*** into local variables
  - Basic scheme:
    - ***Load:*** *t1 = a[i], t2 = b[i+1], ….*
    - ***Compute:*** *t4 = t1 * t2; ….*
    - ***Store:*** *a[i] = t12, b[i+1] = t7, …*

More Difficult Example

- Matrix multiplication: C = A*B + C

```
c = (double *) calloc(sizeof(double), n*n);

/* Multiply n x n matrices a and b  */
void mmm(double *a, double *b, double *c, int n) {
    int i, j, k;
    for (i = 0; i < n; i++)
        for (j = 0; j < n; j++)
            for (k = 0; k < n; k++)
                c[i*n+j] += a[i*n + k]*b[k*n + j];
}
```

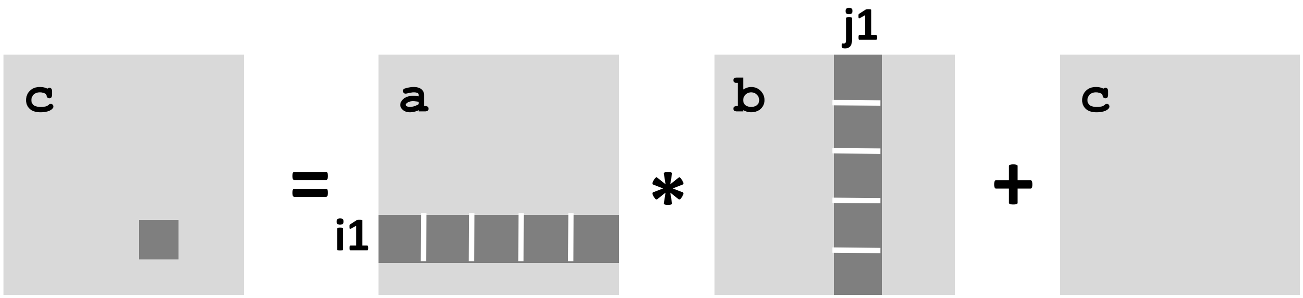


- Which array elements are reused?
- All of them! *But how to take advantage?*

Step 1: Blocking (Here: 2 x 2)

- **Blocking, also called tiling = partial unrolling + loop exchange**
  - Assumes associativity (= compiler will not do it)

```
c = (double *) calloc(sizeof(double), n*n);

/* Multiply n x n matrices a and b  */
void mmm(double *a, double *b, double *c, int n) {
    int i, j, k;
    for (i = 0; i < n; i+=2)
        for (j = 0; j < n; j+=2)
            for (k = 0; k < n; k+=2)
                for (i1 = i; i1 < i+2; i1++)
                    for (j1 = j; j1 < j+2; j1++)
                        for (k1 = k; k1 < k+2; k1++)
                            c[i1*n+j1] += a[i1*n + k1]*b[k1*n + j1];
}
```



Step 2: Unrolling Inner Loops

```
c = (double *) calloc(sizeof(double), n*n);

/* Multiply n x n matrices a and b  */
void mmm(double *a, double *b, double *c, int n) {
    int i, j, k;
    for (i = 0; i < n; i+=2)
        for (j = 0; j < n; j+=2)
            for (k = 0; k < n; k+=2)
                <body>
}
```

```
<body>
c[i*n + j]         = a[i*n + k]*b[k*n + j] + a[i*n + k+1]*b[(k+1)*n + j]
                     + c[i*n + j]
c[(i+1)*n + j]     = a[(i+1)*n + k]*b[k*n + j] + a[(i+1)*n + k+1]*b[(k+1)*n + j]
                     + c[(i+1)*n + j]
c[i*n + (j+1)]     = a[i*n + k]*b[k*n + (j+1)] + a[i*n + k+1]*b[(k+1)*n + (j+1)]
                     + c[i*n + (j+1)]
c[(i+1)*n + (j+1)] = a[(i+1)*n + k]*b[k*n + (j+1)]
                     + a[(i+1)*n + k+1]*b[(k+1)*n + (j+1)] + c[(i+1)*n + (j+1)]
```

- **Every array element a[…], b[…], c[…] used twice**
- **Now scalar replacement can be applied**
  (so again: loop unrolling is done with a purpose)

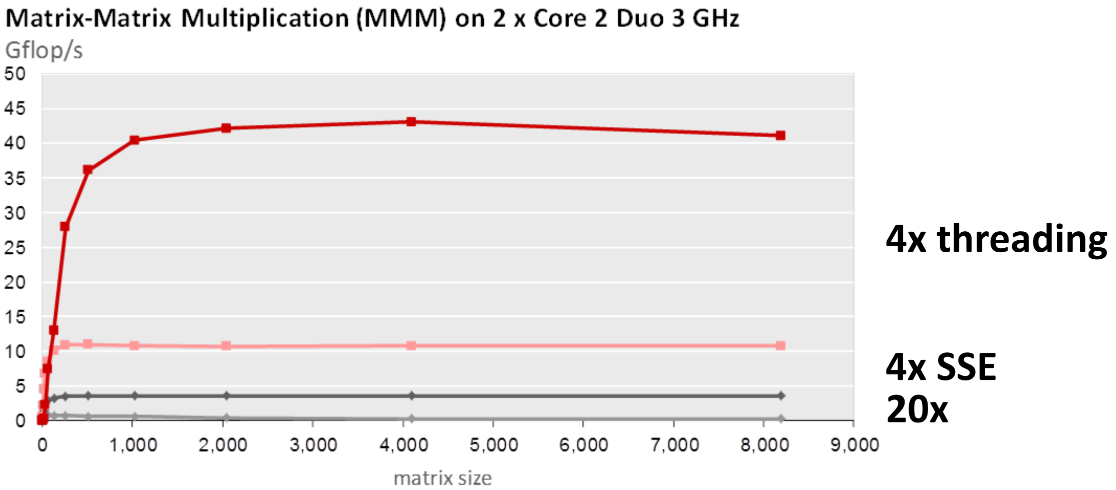
Organization

- Instruction level parallelism (ILP): an example
- **Optimizing compilers and optimization blockers**
  - Overview
  - Removing unnecessary procedure calls
  - Code motion
  - Strength reduction
  - Sharing of common subexpressions
  - Optimization blocker: Procedure calls
  - Optimization blocker: Memory aliasing
  - ***Summary***

Compiler is likely to do that

Summary

- *One can easily loose 10x, 100x in runtime or even more*



- **What matters besides operation count:**
  - Coding style (unnecessary procedure calls, unrolling, reordering, …)
  - Algorithm structure (instruction level parallelism, locality, …)
  - Data representation (complicated structs or simple arrays)

Summary: Optimize at Multiple Levels

- **Algorithm:**
  - Evaluate different algorithm choices
  - Restructuring may be needed (ILP, locality)
- **Data representations:**
  - Careful with overhead of complicated data types
  - Best are arrays
- **Procedures:**
  - Careful with overhead
  - They are black boxes for the compiler
- **Loops:**
  - Often need to be restructured (ILP, locality)
  - Unrolling often necessary to enable other optimizations
  - Watch the innermost loop bodies

Numerical Functions

- **Use arrays if possible**
- **Unroll to some extent**
  - To make ILP explicit
  - To enable scalar replacement and hence register allocation for variables that are reused

Organization

- **Benchmarking: Basics**

Section 3.2 in the tutorial *http://spiral.ece.cmu.edu:8080/pub-spiral/abstract.jsp?id=100*

Benchmarking

- ***First:*** **Verify your code!**
- **Measure runtime in seconds for a set of relevant input sizes**
- **Determine performance [flop/s]**
  - Assumes negligible number of other ops (division, sin, cos, …)
  - Needs arithmetic cost:
    - *Obtained statically (cost analysis since you understand the algorithm)*
    - *or dynamically (tool that counts, or replace ops by counters through macros)*
  - Compare to theoretical peak performance
- ***Careful:*** **Different algorithms may have different op count, i.e., best flop/s is not always best runtime**

How to measure runtime?

- **C clock()**
  - process specific, low resolution, very portable
- **gettimeofday**
  - measures wall clock time, higher resolution, somewhat portable
- **Performance counter (e.g., TSC on Pentiums)**
  - measures cycles (i.e., also wall clock time), highest resolution, not portable
- **Careful:**
  - measure only what you want to measure
  - ensure proper machine state
    (e.g., cold or warm cache = input data is or is not in cache)
  - measure enough repetitions
  - check how reproducible; if not reproducible: fix it
- **Getting proper measurements is not easy at all!**

Example: Timing MMM

- Assume `MMM(A,B,C,n)` computes
  `C = C + AB`, `A,B,C` are `nxn` matrices

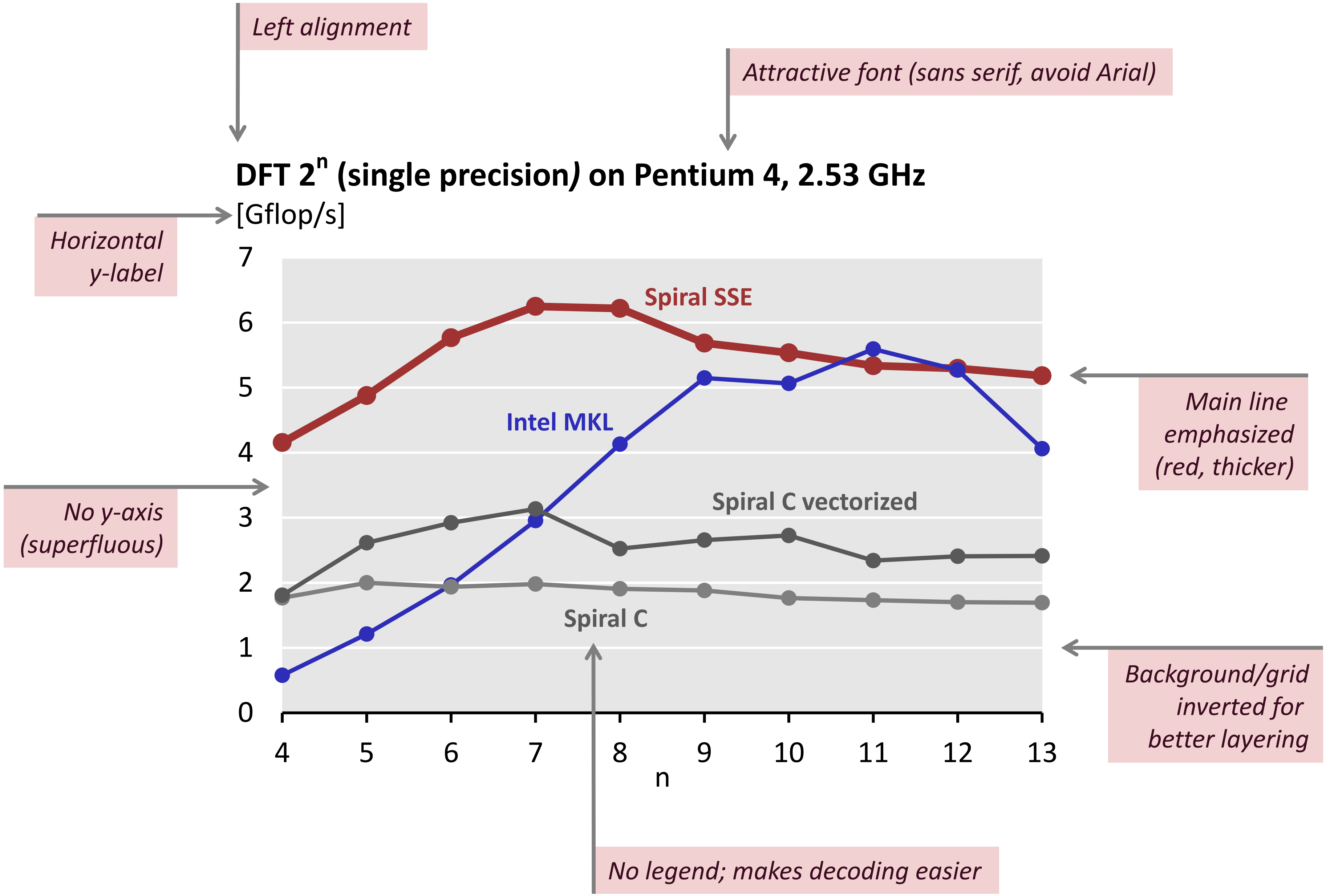
```
double time_MMM(int n)
{ // allocate
  double *A=(double*)malloc(n*n*sizeof(double));
  double *B=(double*)malloc(n*n*sizeof(double));
  double *C=(double*)malloc(n*n*sizeof(double));

  // initialize
  for(int i=0; i<n*n; i++){
    A[i] = B[i] = C[i] = 0.0;
  }

  init_MMM(A,B,C,n); // if needed

  // warm up cache (for warm cache timing)
  MMM(A,B,C,n);

  // time
  ReadTime(t0);
  for(int i=0; i<TIMING_REPETITIONS; i++)
    MMM(A,B,C,n);
  ReadTime(t1);

  // compute runtime
  return (double)((t1-t0)/TIMING_REPETITIONS);
}
```

Problems with Timing

- Too few iterations: inaccurate non-reproducible timing
- Too many iterations: system events interfere
- Machine is under load: produces side effects
- Multiple timings performed on the same machine
- Bad data alignment of input/output vectors: align to multiples of cache line (on Core: address is divisible by 64)
- Time stamp counter (if used) overflows
- Machine was not rebooted for a long time: state of operating system causes problems
- Computation is input data dependent: choose representative input data
- Computation is inplace and data grows until an exception is triggered (computation is done with NaNs)
- You work on a laptop that has dynamic frequency scaling
- Always check whether timings make sense, are reproducible

Benchmarks in Writing

- **Specify platform, compiler and version, compiler flags used**
- **Plot: Very readable**
  - Title, x-label, y-label should be there
  - Fonts large enough
  - Enough contrast (no yellow on white please)
  - Proper number format
    - ***No:*** *13.254687;* ***yes:*** *13.25*
    - ***No:*** *2.0345e-05 s;* ***yes:*** *20.3 μs*
    - ***No:*** *100000 B;* ***maybe:*** *100,000 B;* ***yes:*** *100 KB*


Left alignment
Attractive font (sans serif, avoid Arial)
DFT 2^n (single precision) on Pentium 4, 2.53 GHz
[Gflop/s]
Horizontal y-label
7
6
5
4
3
2
1
0
Spiral SSE
Intel MKL
Spiral C vectorized
Spiral C
Main line emphasized (red, thicker)
No y-axis (superfluous)
Background/grid inverted for better layering
4
5
6
7
8
9
10
11
12
13
n
No legend; makes decoding easier