High Performance and Parallel Computing (MA 6241)

Lecture 4

Program Restructuring for Performance

Ulrich Rüde

Lehrstuhl für Simulation
Universität Erlangen-Nürnberg

visiting Professor at
Department of Mathematics
National University of Singapore

Contact Details

- Course now on IVLE at NUS
- but to contact me it is better to use my German e.mail address
- mail to

ulrich.ruede@fau.de

Contents of High Performance and Parallel Computing

- Introduction
  - Computational Science and Engineering
  - High Performance Computing
- Computer Architecture
  - memory hierarchy
  - pipeline
  - instruction level parallelism
  - multi-core systems
  - parallel clusters
- Efficient Programming
  - computational complexity
  - efficient coding
  - architecture aware programming
  - shared memory parallel programming (OpenMP)
  - distributed memory parallel programming (MPI)
  - parallel programming with Graphics Cards

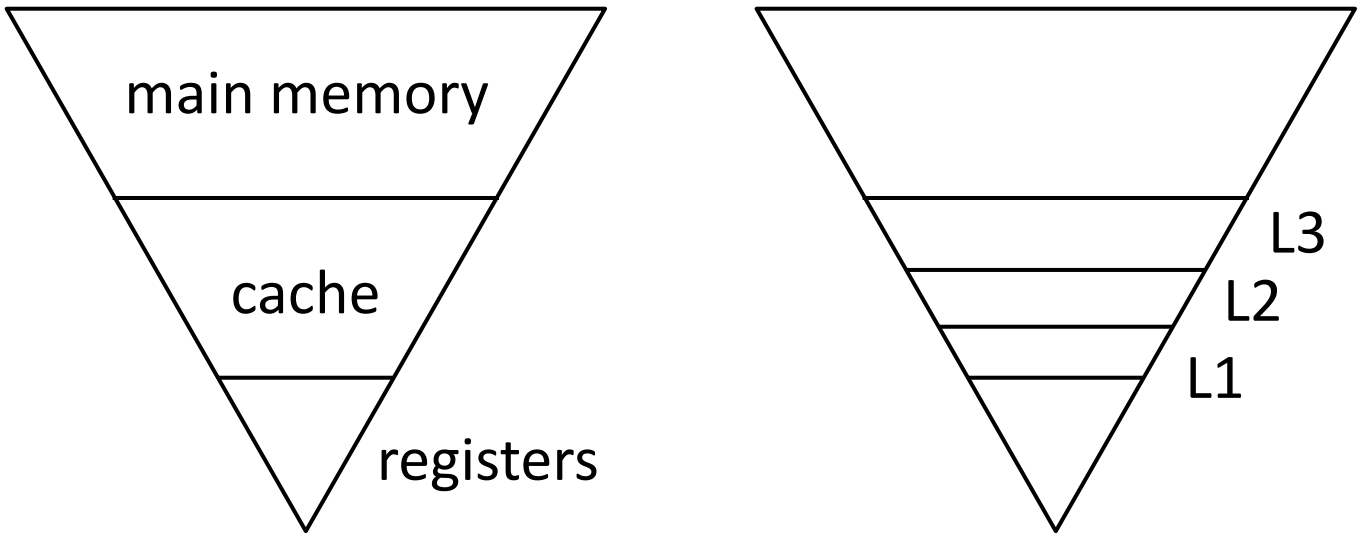
Homework 2 (20%)

1. Your task is to implement a matrix-matrix multiplication
   $C = A \cdot B$, where A, B, and C are square matrices.
2. Feel free to use every known algorithm and programming technique to decrease the single-core runtime of your program as long as you adhere to the following guidelines. All three matrices are
   a) represented as a linearized one-dimensional array with adjacent elements.
   b) passed to your multiplication routine in a row-major format. Meaning: it is not allowed to store one of the input matrices in a transposed layout. However, the computation of the transpose is allowed to be a part of the multiplication routine itself such that the matrix is transposed after the start of the time measurement.
   c) Make sure you use double precision floating-point operations for your multiplication. The use of threads is prohibited.
3. Measure run times for matrix sizes of 1000×1000, 1500x1500, 2000×2000, 3000×3000, 4000×4000, 5000×5000,
4. Compare your run times with that of a highly optimized professional routine (eg. dgemm from BLAS level 3)
5. Summarize your findings on p pages with $p \leqq 2$
   1. due date Feb 16, 2015.
   2. submit as pdf file by e-mail to ulrich.ruede@fau.de

Review and Summary
Processors
Memory Architecture

Memory Hierarchy - Caches

- Use a memory hierarchy in order to decrease the memory gap!
- Cache(s): small but fast memory between main memory (large & slow) and CPU registers (extremely small & fast)



- The cache, like the main memory, is organized in blocks/lines.

Memory Organization (1)

- Registers, typically 32-128 double precision
- Load-Store architecture
- Explicit use of registers: assembly language(?) or register attribute in C, but often ignored by compiler.
- Register allocation algorithms are quite good in state of the art compilers (but not perfect)
- More registers in future CPUs? (128 in Itanium)
- Latency for decoding
- Instruction set (register windows)
- Register allocation more complex, compile times get longer

Cache Types

- Primary cache (L1)
  - Split into data and instruction cache
  - Typical size: 32 – 128 KBytes
  - Intel Core i7 3770: data & instruction cache both 4 x 32 KBytes
- Secondary cache (L2)
  - Unified cache (data and instruction together in the same cache)
  - Typical size: 512 KBytes – 2 MBytes
  - Intel Core i7 3770: 4 x 256 KBytes
- Today often also L3 cache
  - Unified cache, size typically in the range of a few MBytes
  - Intel Core i7 3770: 8 Mbytes (shared between all cores)

Memory Access

If a computer program requires access to a certain data item, different actions may get triggered:

- If the data item is already in the cache: **cache hit** → The data is read from cache.
- Otherwise: **cache miss** → The data first must be fetched from a higher cache level or the main memory.
- If all cache entries are occupied, old items are **evicted** from the cache and replaced by the new data.

Caches

- Fast (but small) extra memory
  - holding identical copies of main momory
  - lower latency
  - higher bandwidth
  - ususally several levels (2 or 3)
  - same principle as virtual memory
- Memory request satisfied from
  - fast cache, if the data is stored there: ***cache hit***
  - or from slow main memory, if the data is not stored in the cache: ***cache miss***
- Issues:
  - Uniqueness and transparency of addressing
  - Finding a working set
  - Data consistency with main memory

Associativity

1. direct mapped (associativity 1): Each main memoy word can be stored in one (and only) one location in the cache.
2. (fully) associative: A main memory word can be stored in any location in the cache
3. set associative (associativity k, typically k=2,4,8,...): Each main memory word can be stored in one of k places in the cache.

- 1 and 3 give rises to conflict misses.
- Direct mapped caches are faster, fully associative caches too expensive and slow (if reasonably large). Set-associative caches are a compromise.

Eviction Strategies

Associative caches need an eviction strategy:
Which block is evicted if the cache or the set is fully occupied?

Strategies:

- Cyclic / first in first out (**FIFO**): The oldest block is evicted.
- Least recently used (**LRU**): The block which was not read for the longest period of time is evicted.
- Least frequently used (**LFU**)
- A **random** block is evicted.
  - Requires minimal hardware effort → fast!
  - In practice only slightly worse than the other strategies!
    (→ predicting the future is difficult …)

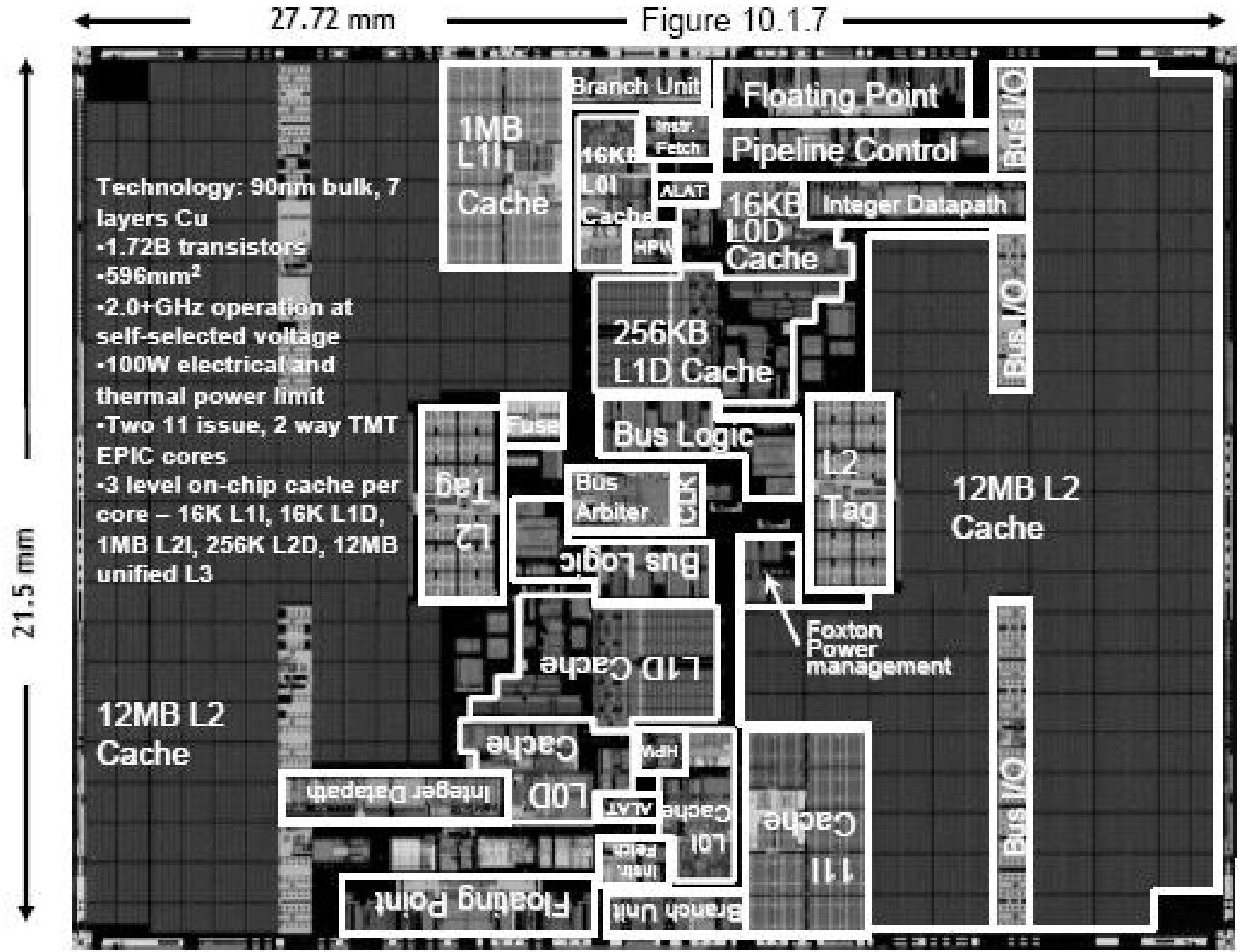
Classification of cache misses

- Compulsory (also called „cold start“ misses): For the first access to a memory location, there is an unavoidable cache miss
- Capacity: The cache is too small to hold all data items. data lines are overwritten according to the replacement policy.
- Conflict: In case of a direct mapped cache or a set-associative cache, it may not be possible to use the full cache size when too many items are mapped to the same set of slots in the cache.

- Good Info on Caches in Wipipedia:
  https://en.wikipedia.org/wiki/CPU_cache

Pipeline Techniques

- a form of on-chip parallelism
- break operation in several steps
  --> high clock rate
- dependencies: result is not finished but already needed for other operations
- (conditional) branches --> branch prediction
  - and/or speculative execution
- pipeline stalls


27.72 mm
Figure 10.1.7
21.5 mm
Technology: 90nm bulk, 7 layers Cu
•1.72B transistors
•596mm²
•2.0+GHz operation at self-selected voltage
•100W electrical and thermal power limit
•Two 11 issue, 2 way TMT EPIC cores
•3 level on-chip cache per core – 16K L1I, 16K L1D, 1MB L2I, 256K L2D, 12MB unified L3
1MB L1I Cache
Branch Unit
Floating Point
Instr. Fetch
16KB L0I Cache
Pipeline Control
ALAT
16KB L0D Cache
Integer Datapath
HPW
Bus I/O
256KB L1D Cache
Fuse
Bus Logic
L2 Tag
Bus Arbiter
12MB L2 Cache
Foxton Power management
12MB L2 Cache

Configuration of SuperMUC

Installation Date	2011	2012	2013	2015
Item	**Fat Node Island**	**Thin Node Islands**	**Many Core Island**	**Haswell Islands**
System	BladeCenter HX5	IBM System x iDataPlex		IBM x3550
Processor Types	Westmere-EX Intel Xeon E7-4870 10C	Sandy Bridge-EP Intel Xeon E5-2680 8C	Ivy-Bridge (IvyB) and Intel Xeon Phi 5110P	Intel Haswell Xeon® Processor E5-2697 v3
Nominal Freqeuncy [GHz]	2.4	2.7		2.6
Number of Islands	1	18	1	6
Nodes per Island	205	512	32	512
Processors per Node	4	2	2 (IvyB) 2.6 GHz + 2 Phi 5110P	2
Cores per Processor	10	8	8 (IvyB) + 60 (Phi)	14
Cores per Node	40	16	16 (host) + 120 (Phi)	28
Logical CPUs per Node (Hyperthreading)	80	32	32 (host) + 480 (Phi)	56
Total Number of nodes	205	9216	32	3096
Total Number of cores	8200	147,456	4352	74,304
Peak Performance [PFlop/s]	0.078	3.185	0.064 (Phi)	3.1
Linpack Performance [PFlop/s]	0.065	2.897	n.a.	
Total size of memory [TByte]	52	288	2.56	192

Program Restructuring for Performance

Data layout

- Access memory in order. For a two dimensional matrix

```
double a[n][m];
```

- the loops should be such that

```
for(i ...)
for(j....)
        a[i][j]
```

(For FORTRAN it must be the other way round)

Other data layout example:

Three vectors accessed together

```
double a[n], b[n], c[n];
```

can often be handled more efficiently by using

```
double abc[n][3];
```

(In FORTRAN again indices permuted)

What compilers can do

Common subexpression elimination

```
s1= a+b+c
s2= a+b-c
```

can be converted to

```
t=  a+b
s1= t+c
s2= t-c
```

What compilers can do

Problem: Trick does not work (without compiler option „-fast“ or equivalent) if e.g. written as

```
s1= a+c+b
s2= a-c+b
```

since associativity does not hold for floating point arithmetic and so the compiler cannot exchange

```
a+c+b
```

with

```
a+b+c
```

unless explicitly permitted (this is unclear in C?).

In a laguage with clean floating point handling the optimization should be prohibited (unless explictly enabled) since the results may differ significantly.

Performance Optimization

A word of advice:

- Before starting hardware related performance optimizations, always choose the **best algorithm** for solving your problem!
  - Naïve matrix-matrix multiplication: $O(N^3) = O(N^{\log_2 8})$
  - Strassen algorithm for matrix multiplication: $O(N^{\log_2 7})$
- In theory, regardless of its constant overhead, an algorithm with a better asymptotic complexity will always solve a given problem in less time – provided the input is large enough.
  - Small matrices → naïve multiplication
  - Large matrices → Strassen algorithm

Cache Aware Program Optimization

The underlying idea:

“A cache-aware algorithm is designed to minimize the movement of memory pages in and out of the processor's on-chip memory cache. The idea is to avoid what's called "cache misses," which cause the processor to stall while it loads data from RAM into the processor cache.” – Jim Mischel

(from: http://stackoverflow.com/questions/473137/a-simple-example-of-a-cache-aware-algorithm)

Temporal Locality

- A common problem:
  - Data is pulled into the cache and not used again for some period of time.
  - It gets evicted.
  - When it is accessed again, it must be reloaded into the cache.
- The solution:
  - Optimize for temporal locality!
  - If some data is accessed multiple times, these accesses should happen within short periods of time – thereby preventing eviction.

- If data is transferred from main memory to the cache, always **cache lines** are transferred, never just single data items.
- Cache line: multiple data items that reside besides each other in the main memory
- Optimizing for spatial locality:
  - If you have to access multiple data items, try to reorder these accesses such that data items that reside in close proximity in main memory are **accessed consecutively** (stride-one / stride-1 memory access).
  - After accessing the first data item, all data items that follow are already in cache!

Spatial Locality

Instruction Level Parallelism

```
program nrm1
    real a(n)
    tt= 0d0

    do j=1,n
        tt= tt+ a(j)*a(j)
    enddo
    print *, tt
end
```

```
program nrm2
    real a(n)
    tt1= 0d0
    tt2= 0d0
    do j=1,n,2
        tt1= tt1+ a(j)*a(j)
        tt2= tt2+ a(j+1)*a(j+1)
    enddo
    tt= tt1+tt2
    print *,tt
end
```

Can this be done automatically?

Change in rounding errors?

Instruction Level Parallelism

```
program nrm3
   real a(n)
   tt= 0d0

   do j=1,n,2
        tt= tt+ a(j)  *a(j)
        tt= tt+ a(j+1)*a(j+1)
   enddo
   print *, tt
end
```

This code could be obtained by simple loop unrolling automatically and it yields bitwise the same result as the original one. However, this code also does not exploit the inherent parallelism, since the access to `tt` becomes a bottleneck.

Loop unrolling by factor 4 or 8 typical.

What is missing: Correct treatment of loop end (loop trailer), especially when `n` is not a multiple of the unroll parameter.

Software pipelining

Consider dependencies on loads:

```
t1= a(1)*a(1)
a1= a(2)
do j=3,n-1,2
        a2= a(j)
        t1= t1+ a1*a1

        a1= a(j+1)
        t2= t2+ a2*a2
enddo
t1= t1+t2+a1*a1
```

- Operations are grouped such that they can be executed together
- Software pipelining is less important on out-of-order processors.
- Software pipelining often done by optimizing compiler, but not always successfully.

Software Pipelining for *daxpy* Operation

```
program dxpy
   real a(n)

   do i=1,n
        a(i)= a(i) + alpha*b(i)
   enddo
end
```

```
do i=1,n-11-12
       a(i+0)= s0
       t8= t8*alpha
       t0= b(i+12)
       s4= s4+t4
       s8= a(i+8)
       .
       .
       .
      a(i+11)= s11
       t7= t7*alpha
       t11= b(i+23)
       s3= s3+t3
       s7= a(i+19)
enddo
```

The softare-pipelined code on the right obtains 550 Mflop instead of 120 on an EV6 (a classical in-order processor - now obsolete). Similar techniques were effective on Cell or Itanium or energy saving in-order processors

Improving the ratio of floating point operations to memory accesses

```
program mlt
   do i=1,n1
         t= 0.d0
         do j=1,n2
            t= t+a(j,i)*x(j)
         enddo
         y(i)= t
   enddo
end
```

- Correct memory access in inner loop (FORTRAN memory layout convention!)
- Use variable t (will be allocated in a register by the compiler) to avoid too many store-operations.
- Each element of a(j,i) read n times!
- Each element of x(j) read n times!
- In the innermost loop, 2 loads are performed and 2 flops.

Multiplies transposed matrix `a(i,j)` with vector `x`.

Floating Point Optimization

Improving the ratio of floating point operations to memory accesses

```
program mlt
   do i=1,n1-3,2
      t1= 0.d0
      t2= 0.d0
      do j=1,n2-3,2
         t1= t1+a(j+0,i+0)*x(j+0)
               +a(j+1,i+0)*x(j+1)
         t2= t2+a(j+0,i+1)*x(j+0)
               +a(j+1,i+1)*x(j+1)

         enddo
      y(i+0)= t1
      y(i+1)= t2

   enddo

end
```

- Both loops unrolled by 2, in practice unrolling by 4 probably better.
- In innermost loop
  6 array elements are loaded,
  8 flops are performed
- How many loads vs. flops when unrolling by 4?

Loop Fusion

- Very simple basic principle (better utilization of registers):

```
for( int i = 0; i < n; ++i )
  b[i] = a[i] + a[i];

for( int i = 0; i < n; ++i )
  c[i] = a[i] * a[i];
```

should be transformed into:

```
for( int i = 0; i < n; ++i ) {
  b[i] = a[i] + a[i];
  c[i] = a[i] * a[i];
}
```

Running out of registers

- Running out of registers
- loop unrolling good for performance
- less loads per flop
- fewer test/jump instructions
- optimizing compilers do automatic loop unrolling
- special compiler options
- inner loops become more complicated
- register spills may occur
- hand-unrolling may still be helpful
- compiler may undo hand-unrolling
- register attribute (in C) mostly ignored
- 32 floating point registers typical
- register renaming
- special registers

Loop unrolling by hand

```
program nrm1
do j=1,n
do i=1,n
   sum= sum+ x(i,j)*(i**2+j**2)
enddo
enddo
```

Automatic unrolling for this program does not produce optimal results.

Instead: consider hand-unrolled program on next slide.

Conversion form int to double may be expensive.

Optimized program

```
sum1= sum2= sum3= sum4= 0d0
rj=-1.d0;  sj= 0.d0
do j=1,n-1,2
   rj= rj+2.d0; rj2= rj*rj
   sj= sj+2.d0; sj2= sj*sj
   ri= -1.d0; si= 0.d0
   do i=1,n-1,2
          ri= ri+2.d0;      si= si+2.d0
          tt= rj2+ri*ri;    ss= rj2+si*si
          uu= sj2+ri*ri;    vv= sj2+si*si
          sum1= sum1+x(i,j)     *tt
          sum2= sum2+x(i+1,j)   *ss
          sum3= sum3+x(i,j+1)   *uu
          sum4= sum4+x(i+1,j+1)*vv
   enddo
enddo
sum= sum1+sum2+sum3+sum4
```

Loop unrolling overhead

Special code needed at loop exit when # of iterations is no multiple of unroll-depth.

This may be expensive and dominate the benefit when unrolling loops that have only a modest number of iterations.

Complete unrolling

Aliasing

Arrays (or other data) that refer to same memory.

FORTRAN forbids aliasing, C permits aliasing $\Rightarrow$

one important reason why FORTRAN compilers may produce faster code than C compilers.

Aliasing Example

```
subroutine sub(n, a,b,c, sum)
double precision a(n), b(n), c(n)
sum= 0d0
do i=1,n
   a(i)= b(i)+ 2.0d0*c(i)
   sum= sum+b(i)
enddo
```

FORTRAN rule: Two variables cannot be aliased, when one or both of them are modified in the subroutine.

Correct call:

```
double precision a(n), b(n), c(n), sum
call sub(n,a,b,c,sum)
```

Incorrect call (forbidden). The result is undefined, i.e. compiler may produce any result:

```
double precision a(n), b(n), c(n)
call sub(n,a,a,c,sum)
```

Why should aliasing be forbidden?

Consider a compiler attempting to software-pipeline the previous subroutine:

```
tb= b(1)
tc= c(1)
do i=1,n-1
   ta= tb+2.d0*tc
   tc= c(i+1)

   sum= sum+tb
   tb= b(i+1)

   a(i)= ta
enddo
i=    n
ta=   tb+2.0*tc
sum=  sum+tb
a(i)= ta
```

- Assume that each block of ops can be executed in 1 cycle
- latency 1 cycle for load
- latency 2 cycles for a flop
- Program produces „wrong“ results when used with second call, since `a ≡ b`.

Why should aliasing be forbidden?

Consider version that is correct with aliasing:

```
do i=1,n
   LOAD tc= c(i)
   LOAD tb= b(i)
   ta= tb*2d0*tc
   NOOP
   STORE a(i)= ta
   LOAD sum
   LOAD tb= b(i)
   sum= sum+ tb
   STORE sum
enddo
```

- `NOOP` indicates CPU waiting because of dependencies
- explicit `LOAD/STORE`
- couldn't we save `LOAD/STORE` of sum?

C/C++ optimizers are much more difficult to write, sometimes they cannot produce as fast code as FORTRAN

Aliasing

- Aliasing forbidden in FORTRAN (result undefined when used)
- Aliasing in C/C++ legal: Compiler must produce conservative code
- More complicated aliasing could occur, e.g. `a(i)` with `a(i+2)`.
- C/C++ compilers have a key word *restrict* or a compiler option *-noalias*.
- Programmer help for C-compilers:

```
double precision ah, bh, ch
sum= 0d0
do i=1,n
   ah= a(i), bh= b(i), ch= c(i)
   ah= bh+ 2.0d0*ch
   sum= sum+bh
   a(i)= ah
enddo
```

Another aliasing example

```
subroutine sub(n,a,b,ind)
   double precision a(n), b(n)
   integer ind(n)
   do i= 1,n
            b(ind(i))= 1.0d0 + 2.0*b(ind(i)) + 3.0*a(i)
   enddo
   return
end subroutine
```

Loop iterations are not independent (and cannot be easily pipelined) when `ind(i) = ind(i+1)`.

Some compilers have options/directives to permit aggressive optimization when the programmer guarantees that `ind(i)` is an injective mapping.

Pointers and indirect addressing are dangerous wrt. aliasing.

Subroutine calling overhead

- Subroutines (functions) are very important for structured, modular programming.
- Subroutine calls are not cheap (on the order of up to 100 cycles).
- Passing value arguments (copying data) can be extremely expensive, when used inappropriately.
- Passing reference arguments (as in FORTRAN) may be dangerous (from a point of view of correct software).
- Reference arguments (as in C++) with const declaration.
- Generally, in *tight loops*, no subroutine calls should be used

Inlining

- inline declaration in C++, or done automatically by compiler.
- Macros in C (or any other language)

```
#define sqre(a)    (a)*(a)
```

- What can go wrong:

```
sqre(x+y)     ---->     x+y*x+y

sqre(f(x))    ---->     f(x) * f(x)
```

- What if `f` has side effects?

- What if `f` has no side effects, but the compiler cannot deduce that?