Masterpraktikum Scientific Computing

High-Performance Computing

Michael Bader
Alexander Heinecke
Alexander Breuer

Technische Universität München, Germany

Outline

Logins

Levels of Parallelism

Single Processor Systems
- Von-Neumann-Principle
- Accelerating von-Neumann: Pipelining
- Loop Optimizations
- Pipeline-Conflicts
- Dependency Analysis

Caches

Vectorcomputers
- Architecture of a Vectorcomputer
- Example: SSE-Extension of Intel Core 2
- Dependency Analysis and Vectorization

Logins LRZ

- logins: one login per participant
- change password through: `https://idportal.lrz.de/r/entry.pl`: *Passwort ändern* on left side, type old one, repeat your new one two times and hit *Passwort ändern* afterwards.
- ssh xyz@mac-login-amd.tum-mac.cos.lrz.de or xyz@mac-login-intel.tum-mac.cos.lrz.de, within the MWN (VPN client or computing hall)
- read the motd carefully
- module system: module load, list, unload, info
- **Documentation** `http://www.lrz.de/services/compute/linux-cluster/intro/`

Outline

Logins

Levels of Parallelism

Single Processor Systems
- Von-Neumann-Principle
- Accelerating von-Neumann: Pipelining
- Loop Optimizations
- Pipeline-Conflicts
- Dependency Analysis

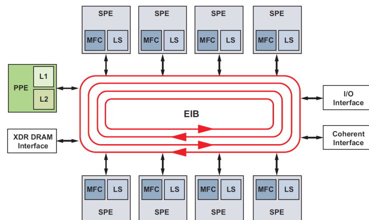
Caches

Vectorcomputers
- Architecture of a Vectorcomputer
- Example: SSE-Extension of Intel Core 2
- Dependency Analysis and Vectorization

Levels of Parallelism

Parallel applications can be classified according to their granularity of parallelism. Different kinds of hardware support these different levels of parallelism.

- intra-instruction level
  - vector-computer (e.g. NEC-SX, Cray-SV)
  - SSE, AVX, Intel MIC
  - Cell, GPUs
  - ...



- instruction level
  - super-scalar architecture (e.g. Workstation/PC)
  - VLIW architecture (very long instruktion word) (e.g. Itanium, AMD GPUs)
  - pipelining
  - ...

ILP – **I**nstruction **L**evel **P**arallelism

- thread level
  - Shared-Memory-Machine (e.g. SUN-UltraSparc, SGI-Altix)
  - . . .
- process level
  - Distributed-Memory-Machine (Cray-T3E, IBM-SP)
  - . . .
- application level
  - Workstation-Cluster (SuperMUC)
  - Grid-Computer
  - . . .

Classification according to Flynn

- instruction flow and / vs. data flow
- handling several instructions simultaneously
- handling several data elements simultaneously

SISD scalar machine (486)	**SIMD** vector-machine (since Pentium MMX!) GPGPU (VLIW)
MISD	**MIMD** GPGPU (SIMT)

Outline

Logins

Levels of Parallelism

Single Processor Systems
- Von-Neumann-Principle
- Accelerating von-Neumann: Pipelining
- Loop Optimizations
- Pipeline-Conflicts
- Dependency Analysis

Caches

Vectorcomputers
- Architecture of a Vectorcomputer
- Example: SSE-Extension of Intel Core 2
- Dependency Analysis and Vectorization

Von-Neumann-Principle

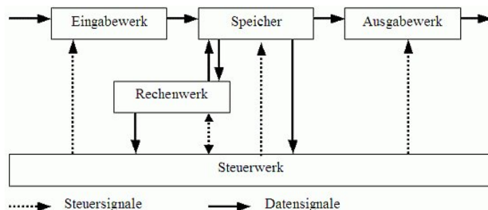


Figure: Von-Neumann-machine

IF:	instruction fetch
DEC:	instruction decode register fetch
EXEC:	execution effective address branch output
MEM:	memory access branch comptetion
WB:	write back

Figure: Stages of an instruction cycle

Pipelining

- instruction pipelining: overlapping of independent instructions
- increasing instruction throughout
- a pipeline with k stages retires, after a certain preload-time, starting with cycle k one instruction per cycle!

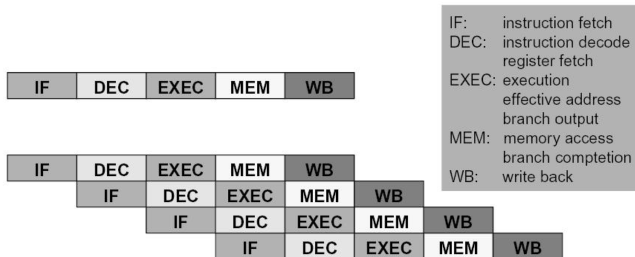


Figure: execution with and without pipelining

Arithmetic Pipelining

Example: Floatingpoint-Addition in four steps

- loading operands
- compare exponents, shift mantissa
- add
- normalization and write back to registers

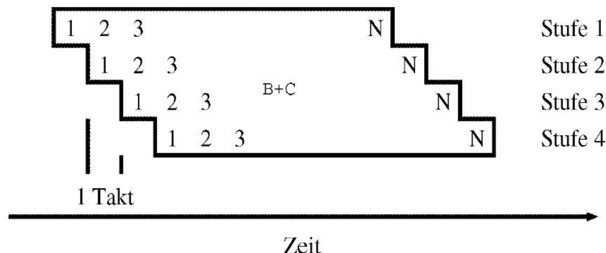


Figure: vector operation pipeline

Pipelining Optimizations

- loop unrolling

```
for(i=0;i<3;i++) {
   for(k=0;k<m;k++) {
      c[i,k]=a[i,k]*b[i,k];
   }
}
```

```
for(k=0;k<m;k++) {
    c[0,k]=a[0,k]*b[0,k];
    c[1,k]=a[1,k]*b[1,k];
    c[2,k]=a[2,k]*b[2,k];
}
```

- loop fusion

```
for(i=0;i<n;i++) {
    d[i]=d[i-1]*e[i];
}

for(i=0;i<n;i++) {
    f[i]=f[i]+c[i-1];
}
```

```
for(i=0;i<n;i++) {
    d[i]=d[i-1]*e[i];
    f[i]=f[i]+c[i-1];
}
```

- for further details, see **Intel C++ Compiler User Guide**

Pipeline-Conflicts

- **structure-conflict:** parallel execution is prohibited due to hardware limitations (execution units, memory ports)
- **branch-conflict:** jumps
  $\Rightarrow$ branch-prediction, speculative execution
- **data-conflict:** conflicting data-accesses ($c = a + b, d = c + e$)

$\Rightarrow$ compilers try to avoid conflicts as best as possible. Hardware plays several tricks in order to avoid conflicts at runtime!

Dependency Analysis

```
S1: A = C - A
S2: A = B - C
S3: B = A + C
```

- True dependence (RAW) (e.g. **S3** to **S2** w.r.t. **A**)
  - **S3** gives a true dependence to **S2**, **S3** δ **S2**,
    if **OUT(S2)** $\cap$ **IN(S3)** $\neq \emptyset$;
    $\Rightarrow$ **S3** uses result of **S2**
- Anti dependence (WAR) (e.g. **S3** to **S2** w.r.t. **B**)
  - **S3** gives a anti dependence to **S2**, **S3** δ^{-1} **S2**,
    if **IN(S2)** $\cap$ **OUT(S3)** $\neq \emptyset$;
    $\Rightarrow$ exchanging **S3** and **S2** will result in using the results of **S3** for **S2**

Dependency Analysis

```
S1: A = C - A
S2: A = B - C
S3: B = A + C
```

- Output dependence (WAW) (e.g. **S2** to **S1** w.r.t. **A**)
  - **S2** and **S1** are an output dependence, **S2** δ^O **S1**, if **OUT(S1)** $\cap$ **OUT(S2)** $\neq \emptyset$;
    $\Rightarrow$ exchanging **S2** and **S1** result in wrong values.

$\Rightarrow$ Dependency Analysis checks if pipelining or vectorization is possible!

Outline

Logins

Levels of Parallelism

Single Processor Systems

- Von-Neumann-Principle
- Accelerating von-Neumann: Pipelining
- Loop Optimizations
- Pipeline-Conflicts
- Dependency Analysis

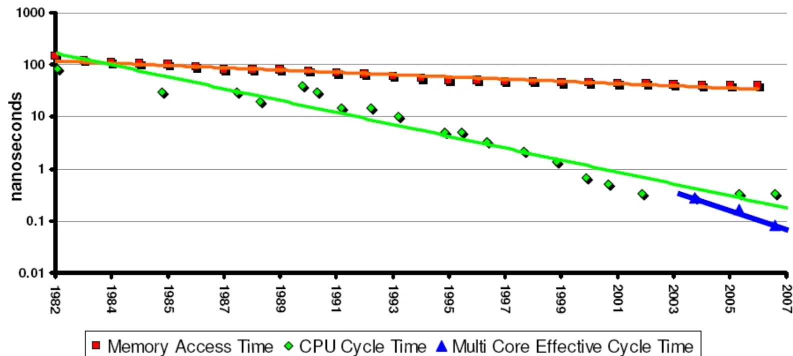
Caches

Vectorcomputers

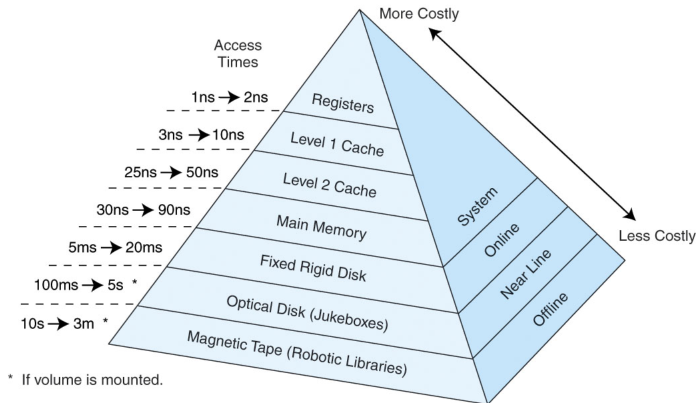
- Architecture of a Vectorcomputer
- Example: SSE-Extension of Intel Core 2
- Dependency Analysis and Vectorization

Caches - Motivation

Processor- vs. Memory-Speed

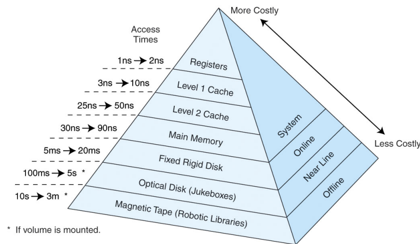


Memory-hierarchy

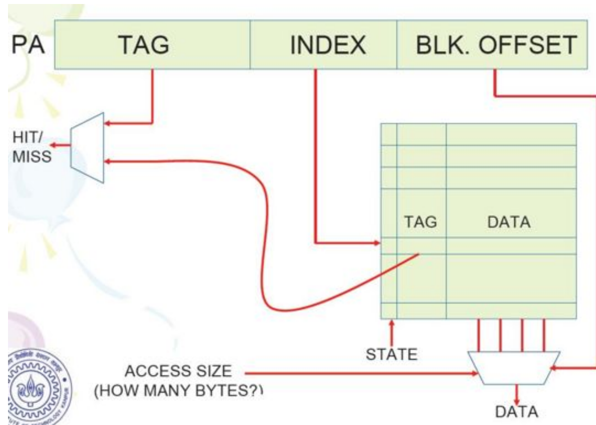


Caches

- transparent buffer memory
- exploiting locality:
  - **temporal locality**: If we access address x, we probably will access address x in near future again.
  - **spatial locality**: If we access address x, we probably will access addresses $x + \delta$ or $x - \delta$, too.



Caches - Structure



Beispiel

- 512 cache-lines a 64 Bytes (= 32KB)
- 32 bit addresses (6 bit Block-Offset, 9 bit Index, 17 bit Tag)

Caches - Examples

- Intel Core i7
  - 32 KB L1-data-cache, 8-way associative
  - 32 KB L1-instruction-cache
  - 256 KB L2-Cache, 8-way associative
  - 8 MB shared L3-Cache, 16-way associative
  - 64 Byte cache-linesize
- AMD Phenom II X4
  - 64 KB L1-data-cache, 2-way associative
  - 64 KB L1-instruction-cache
  - 512 KB L2 Cache, 16-way associative
  - 6 MB shared L3 Cache, 48-way associative
  - 64 Byte cache-linesize

Caches - How to?

- mapping-strategy (associativity)
  - fully associative cache
  - direct mapped cache
  - set associative cache
- replacement strategy
  - First In First Out (FIFO)
  - Least Recently Used (LRU)
- update strategy
  - Write Through
  - Write Back
- cache-coherency (organize data validity across non-shared caches)

Cache-Misses

- **cold misses**: first access of data at address x (not avoidable!)
- **capacity misses**: evict of data at address x due to limited size of cache(-level)
- **conflict misses**: evict of data at address x due to associativity of cache(-level)

Cache-Blocking

- example matrix-multiplikation

```
for(i = 0; i < n; i++)
 for(j = 0; j < n; j++)
   for(k = 0; k < n; k++)
     c[i,j] += a[i,k] * b[k,j];
```

Cache-Blocking

- example matrix-multiplikation

```
for(i = 0; i < n; i++)
  for(j = 0; j < n; j++)
    for(k = 0; k < n; k++)
      c[i,j] += a[i,k] * b[k,j];
```

  - **Register-Blocking**: bandwidth to L1-cache is to small for moving 3 operands per cycle (e.g. Intel Sandy Bridge 48 bytes per cycle, but we would need 96 bytes!)
  - **Cache-Blocking**: bandwidth of main memory is orders of magnitudes slower than caches (200 cycles vs. 10 cycles)
  - **TLB-Blocking**: change data structure in order to avoid DTLB misses, usage of huge pages

Outline

Logins

Levels of Parallelism

Single Processor Systems
- Von-Neumann-Principle
- Accelerating von-Neumann: Pipelining
- Loop Optimizations
- Pipeline-Conflicts
- Dependency Analysis

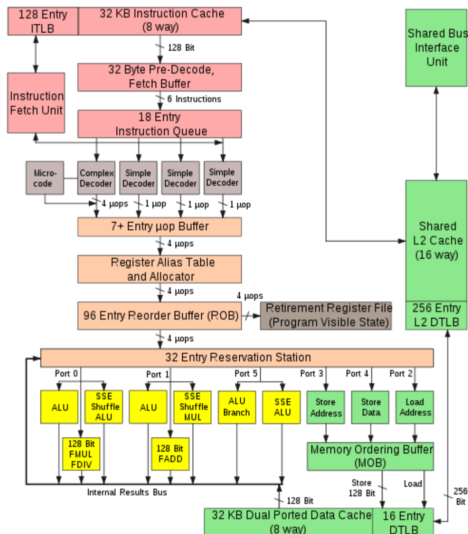
Caches

Vectorcomputers
- Architecture of a Vectorcomputer
- Example: SSE-Extension of Intel Core 2
- Dependency Analysis and Vectorization

Architecture of a Vectorcomputer

- In the 80s and early 90s: A machine with already discussed **pipelined** execution units which are operating on **Arrays** of float-point numbers.
- Today: execution of entire **vector-instructions** on **vector-registers**

Example: SSE-Extension of Intel Core 2



Intel Core 2 Architecture

SSE-Registers

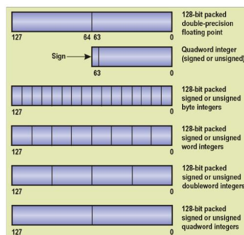


Figure: Streaming SIMD Extensions data types

- Streaming SIMD Extension (SSE)
  - 16 registers (xmm0 to xmm15, each 128 bit wide)
  - 2 double-precision floating point values (each 64 bit)
  - 4 single-precision floating point values (each 32 bit)
  - from 2 64-bit integer values up to 16 8-bit integer values

AVX-Register

- Advanced Vector Extensions (AVX)
  - Since Intel Sandy Bridge or AMD Bulldozer
  - 16 Register (ymm0 to ymm15, je 256 bit wide)
  - 2 double-precision floating point values (each 64 bit)
  - 4 single-precision floating point values (each 32 bit)
  - currently no integer support, will be added with AVX2 (Intel Haswell)
  - full masking support

Auto-Vectorization

```
#pragma vector always
#pragma ivdep
for(i=1;i<n;i++) {
   c[i]=s*c[i];
}
```

Intel Compiler Pragmas for Intel64 Vectorization:

- #pragma vector always ⇒ vectorize in any case (even if compiler detects inefficiencies)
- #pragma ivdep ⇒ disable some dependency checks during compilation
- #pragma simd ⇒ aggressive vectorization, might result in wrong code!

further information (SSE & AVX):

- Intel Architecture Optimization Reference Manual
- Intel C++ Compiler User's Guide

Dependency Analysis and Vectorization

- Loop I

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

no dependency across iterations ⇒ can be optimally vectorized and pipelined

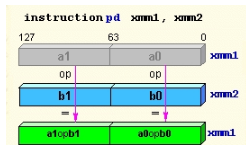
- Loop II

```
for(i=1;i<n-1;i++) {
      c[i]=c[i]*c[i-1];
}
```

Recursion! c_{i-1} is still processed when calculating c_i ⇒ no vectorization is possible

SSE/AVX Intrinsic Functions

- `__m128i; __m128d; //Register-Variablen SSE`
  `__m256i; __m256d; //Register-Variablen AVX`
- `__m128d _mm_`*instr*`_pd(__m128d a, __m128d b)`
  `__m256d _mm_`*instr*`_pd(__m256d a, __m256d b)`
- suffixes:
  - ps,pd → packed single- und double-precision floating point functions
  - ss,sd → scalar single- und double-precision floating point functions
  - ...
- documentation of all function in chapter "Intel C++ Intrinsics Reference" of **Intel C++ Compiler User's Guide**



Example: SSE Intrinsic Functions

```
#include <immintrin.h>

int sse_add(int length, double* a,
                double* b, double* c) {
    int i;
    __m128d t0,t1;

    for (i = 0; i < length; i +=2) {
        // loading 2 64-bit-double values
        t0 = _mm_load_pd(a[i]);
        t1 = _mm_load_pd(b[i]);
        // adding 2 64-bit-double values
        t0 = _mm_add_pd(t0,t1);
        // storing 2 64-bit-double values
        _mm_store_pd(c[i],t0);
    }
}
```

Example: AVX Intrinsic Functions

```
#include <immintrin.h>

int avx_add(int length, double* a,
            double* b, double* c) {
    int i;
    __m256d t0,t1;

    for (i = 0; i < length; i +=4) {
        // loading 4 64-bit-double values
        t0 = _mm256_load_pd(a[i]);
        t1 = _mm256_load_pd(b[i]);
        // adding 4 64-bit-double values
        t0 = _mm256_add_pd(t0,t1);
        // storing 4 64-bit-double values
        _m256m_store_pd(c[i],t0);
    }
}
```