


# High Performance Computing Lecture 1



---

**Alexandros Stamatakis**  
Junior Research Group Leader

**The Exelixis Lab**  
Department of Computer Science  
Technical University of Munich

**[stamatak@cs.tum.edu](mailto:stamatak@cs.tum.edu)**  
**<http://icwww.epfl.ch/~stamatak>**

# Organizational Issues



---

- Lecture 2: Tuesday 28.10 from 16:00-18:00 Room A105
- In this lecture we will do a C crash course due to popular request!
- Course will be classified as belonging to area P: “Prog. und Softwaretechnik”
- Integrate Erasmus students!
- Now the other issue:

# Course Credits

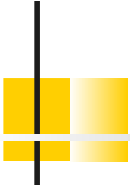
Option 1:  
3SWS with Übung  
and 4 ECTS credits

Option 2:  
2SWS, 3 ECTS  
Prog. Exercises every  
2 weeks

	1. Übung & ...	2. Ohne Übu...	3. C crash ...
florian	OK		OK
Manuel Lindauer		OK	OK
Zsolt Komornik	OK		
Serghei Novojilov	OK		
Sebastian	OK		OK
Michael Würtinger	OK		
Harald Kraft		OK	
Björn Cullmann		OK	OK
Minh Bui		OK	OK
Konstantin		OK	OK
Flo Lindemann	OK		OK
Sebastian Pölsterl	OK		OK
Erika Kalix	OK		OK
Martin Gaißer	OK		OK
Fabian Knopf		OK	
Julian Köpke		OK	OK
Jan		OK	OK
Szymon Seliga	OK		OK
Sebastian Skambraks		OK	OK
Ihr Name	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Anzahl	10	9	14
<input type="button" value="Teilnehmen"/>			

# Questions about homework?

---



# Outline: Next Lectures

---

- Lecture 1 (22.10)
  - loop unrolling
  - Matrix allocation, Lapack & BLAS
  - Gprof & Valgrind usage
  - floating point numbers
  - ILP: instruction level parallelism
- Lecture 2 (28.10)
  - C crash course
- Lecture 3 (05.11)
  - Cache architectures
- Lecture 4 (12.11)
  - Parallel Computer Architectures
  - Measuring Performance
- Lecture 5 (19.11) to be replaced
  - Introduction to Pthreads
  - Open subject
- Lecture 6 (26.11)
  - Report from IEEE/ACM Supercomputing 2009
  - Top 500 list

# Recap



---

- We were talking about loop optimization
- Loop unrolling
- Loop jamming/fusion
- Loop inversion
- Loop vectorization
- Let's continue a bit

# Inlining

```
void doSomething(double *a)
```

```
{  
    *a = *a * exp(*a);  
}
```

```
for(i = 0; i < 1000; i++)
```

```
{  
    doSomething(&a[i]); /* calling this function means transfer of control, allocation of heap  
                        space */  
    b[i] = a[i] * 2.0;  
}
```

```
for(i = 0; i < 1000; i++)
```

```
{  
    a[i] = a[i] * exp(a[i]);  
    b[i] = a[i] * 2.0;  
}
```

Your code might get ugly, alternatively:

```
static void inline doSomething(double *a)
```

```
{  
    *a = *a * exp(*a);  
}
```

# Invariant Computations

```
for(i = 0; i < 1000; i++)
{
    for(j = 0; j < 1000; j++)
        a[i * 1000 + j] = a[i * 1000 + j] * b[j];
}
```

```
for(i = 0; i < 1000; i++)
{
    int i_1000 = i * 1000;
    for(j = 0; j < 1000; j++)
        a[i_1000 + j] = a[i_1000 + j] * b[j];
}
```

or

```
for(i = 0; i < 1000; i++)
{
    double *ptr = &a[i * 1000];
    for(j = 0; j < 1000; j++)
        ptr[j] = ptr[j] * b[j];
}
```

# Loop Stuff

- Move constant/invariant expressions out of loops!
- Avoid conditional expressions in loops
- Replace  $a = x / y$  with  $a = x * 1.0/y$  divisions are expensive
- Avoid calling functions from within loops
- Analyze while do-while and for loop performance
- Avoid recursion if possible (Why?)
- **Suggestion:** play with all these things with compiler optimizations switched off!
- HPC means to play with machines (frickeln)

# C Stuff

- How to store  $n \times n$  arrays?
- `double **a`
  - Alloc `**` first and then `*`
  - Alloc `n * n double` first and assign `**`
- Alloc `n * n` right away and index as linear array
- In C we use rows first by tradition
- In Fortran it is columns first
- Why should we do these things?
- Why do we need this?

# BLAS & Lapack



---

- Implementations
  - GOTO-BLAS
  - ATLAS-BLAS
  - Intel MKL Math Kernel Library
  - AMD AMCL
- Interfaces: by rows or by columns?
- ATLAS has a C interface

# ATLAS: Automatically Tuned Linear Algebra Package

---

- <http://math-atlas.sourceforge.net/>
- It's freely available :-)
- It adapts itself automatically and dynamically to a system
- Fortran 77 **and** C interfaces
  - BLAS routines
  - A couple of Lapack routines

# BLAS: Basic Linear Algebra Subprograms

---

- Level 1 routines
  - Vector-vector ops
- Level 2 routines
  - Vector-matrix ops
- Level 3 routines
  - Matrix-matrix ops
- C-interface:
  - Functions preceded by `cblas_`, e.g., `cblas_dgemm`
  - Arrays need to be contiguous in memory
  - Passed as pointers, **not** pointers of pointers

# C-Interface



---

- Include `cblas.h`
- `CblasRowMajor`: rows first
- `CblasColumnMajor`: columns first
- All operands of a BLAS call need to be in same order!
- Prefixes, e.g., `dgemm`
  - S (Real, single precision)
  - D (double precision)
  - C (complex)
- Think about ordering your arrays before integrating BLAS!
- Atlas can also be obtained as pre-compiled binary
- ... or as Ubuntu package via the package manager

# Further C Stuff

- Reduce function call overhead
  - Reduction will generate less stack traffic
- Define small functions as inline (Windows does not always recognize this)
- Use register modifier, e.g., “register int a;” (not so important any more)
- Use  $x \gg 2$  instead of  $x / 4$
- Use  $x < 1$  instead of  $x * 2$
- Try to avoid floating point arithmetics



# Redundant Operations

---

```
double x = d * (lim / max) * sx;
```

```
double y = d * (lim / max) * sy;
```

```
double depth = d * (lim / max);
```

```
double x = depth * sx;
```

```
double y = depth * sy;
```

This is of course more important in loops!

# Using Gprof

- GNU profiler
- Uses PC sampling technique
- Add the -pg flag to the compiling step!
- Add the -pg flag to the linking step!
- Then just run the program as before
  - It will execute slower though!
- Slowdown for mmult:
  - Less than 1% perhaps not surprisingly  
(why?)
- Then type “gprof mmult”

# Gprof on mmult

Flat profile:

Each sample counts as 0.01 seconds.

% cumulative time	self seconds	self seconds	self calls	total ns/call	total ns/call	name
99.83	156.90	156.90				main
0.17	157.17	0.27	11184640	24.14	24.14	randum
0.00	157.17	0.00	18	0.00	0.00	gettime

% time      the percentage of the total running time of the program used by this function.

.....

# Using Valgrind

- Open source tool to find memory leaks and a lot of other stuff!
- When compiling add `-g` to the compiler flags such that you can see the source lines where valgrind detects errors
- To look for memory leaks type: `valgrind --tool=memcheck --leak-check=yes -v` and then just the executable as you called it so far
- Also works with Pthreads-based programs
- Valgrind **slowdown** for `mmult` **almost factor 10 (188 -> 1451 secs)!**
- Hard to use if problems for very large mallocs occur!
- A comment about `malloc(size_t size);`
  - `size_t` is an unsigned integer or an unsigned long integer depending on the system!
  - It is a good idea to build a wrapper around `malloc` and do a little assertion, e.g., `assert(size > 0) !`

# Valgrind on Mmult

Total time 1451.554931

--13304-- REDIR: 0x40cd4b0 (memset) redirected to 0x4023d50 (memset)

==13304==

==13304== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 13 from 1)

--13304--

--13304-- supp: 13 dl-hack3-1

==13304== malloc/free: in use at exit: 0 bytes in 0 blocks.

==13304== malloc/free: 12,264 allocs, 12,264 frees, 134,264,640 bytes allocated.

**Deliberate bug:** `c[matrixSize][matrixSize-1] = 0.0;`

==14585== Invalid read of size 4

==14585== at 0x80488AC: main (mmult.c:124)

==14585== Address 0x41aa148 is 0 bytes after a block of size 64 alloc'd

==14585== at 0x4022AB8: malloc (vg\_replace\_malloc.c:207)

==14585== by 0x8048625: main (mmult.c:73)

==14585==

==14585== Invalid write of size 8

==14585== at 0x80488BC: main (mmult.c:124)

==14585== Address 0x78 is not stack'd, malloc'd or (recently) free'd

# Double versus single precision

---

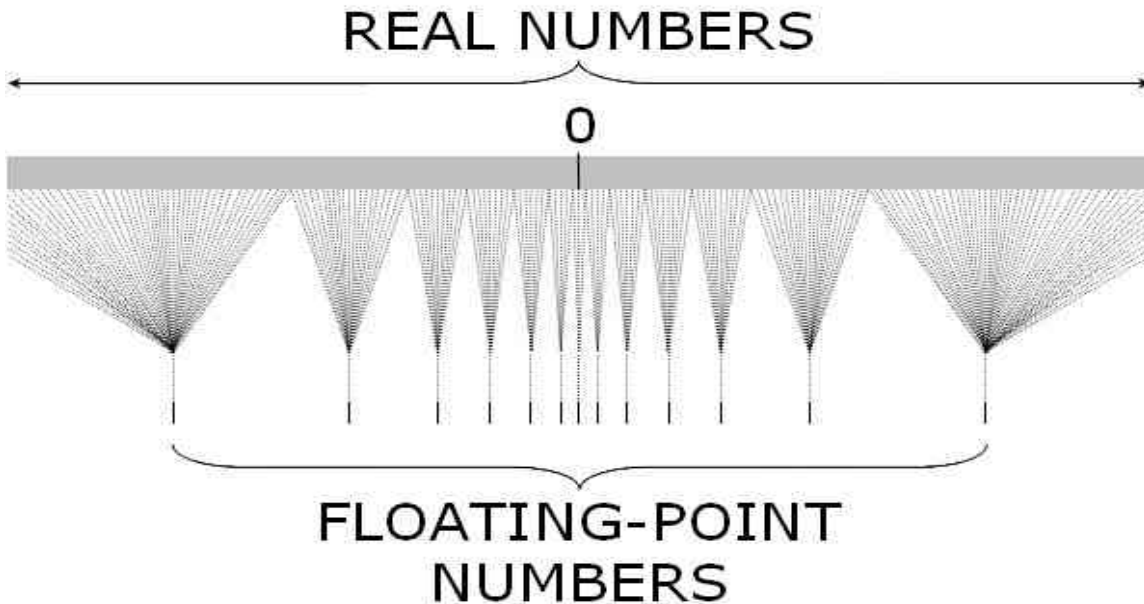
- Single precision arithmetics are faster than double precision arithmetics
- Current GPUs only offer single precision
- IBM Cell/BE & PS III are much faster for single precision (factor  $\approx 20$ ), but IBM is working on improving that
- Single precision may lead to more numerical problems with over-/underflow
- ... but require only half the memory!
- Classic engineering trade off:
  - Precision versus Speed
  - Potential Strategy: switch between phases of single and double precision operations if problem permits!

# Floating Point

- What can we represent in N bits?
  - Unsigned integers:  
0 to  $2^N - 1$
  - Signed Integers (Two's Complement)  
 $-2^{(N-1)}$  to  $2^{(N-1)} - 1$

# What are machine numbers?

- Machine numbers represent a mapping of the infinite real numbers to a finite number of machine values (also called floating point numbers) represented as strings of bits
- floating point numbers have a finite number of bits, e.g., 32, 64, 128 etc.



# Floating Point



---

- Scientific notation:
  - 314.569 is  $3.14569 \times 10^2$
  - This is a normalized number
  - Binary numbers represented analogously:  
 $1.00011 \times 2^3$
- FLT point arithmetics
  - Binary point is not fixed, it floats :-)
  - Format facilitates floating point operation algorithms
  - Increases accuracy leading zeros are eliminated !

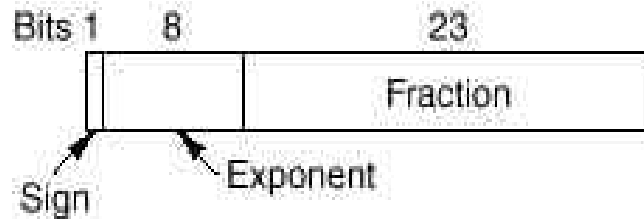
# Floating Point

- Need to represent fraction **f** and exponent **exp**
- **Remember:** fixed limited number of bits available
- Classic engineering trade-off: how many bits for **f** and how many for **exp**?
- High **exp** increases range!
- High **f** increases accuracy!
- Example: MIPS (RISC processor architecture)
  - 1 bit for sign
  - 8 bits for exponent
  - 23 bits for fraction
  - Range:  $2 \times 10^{-38}$  to  $2 \times 10^{38}$

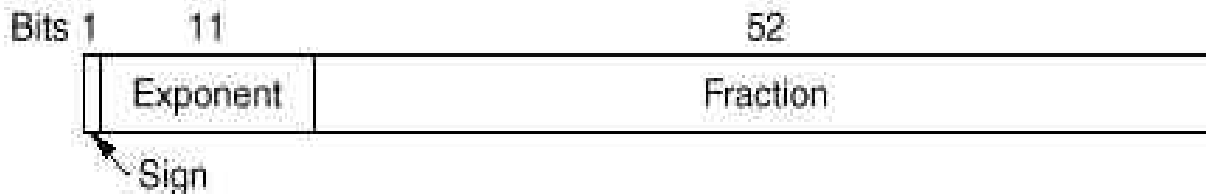
# Floating Point

- Overflow & Underflow
  - Number too small/large for representation
  - e.g., previous example (s1/f23/e8) 40 leading zeros after the binary point :-)
- Double precision
  - From 32 to 64 bits :-)
  - (s1/f52/e11) accuracy-range trade-off
  - Numbers from  $2 \times 10^{-308}$  to  $2 \times 10^{308}$

# IEEE 754



(a)



(b)

**Figure B-4.** IEEE floating-point formats. (a) Single precision. (b) Double precision.

# Floating Point Addition

- $9.999 \times 10^1 + 1.610 \times 10^{-1}$
- First we need to align the exponents
- $1.610 \times 10^{-1} = 0.01610 \times 10^1$
- Step 1
  - Shift the mantissa of the number with the smaller exponent  $n$  bits to the right to align exponents
  - We may lose precision here!
- Step 2
  - Add the fractions/mantissas

# Floating Point Addition

- Step 3
  - Normalize results to leading 1 before the point (leading 1 in binary that is)
  - Shift left or right appropriately
  - $10.015 \times 10^2 = 1.0015 \times 10^3$  (normalized form)
- Step 4
  - We only have a limited number of bits for the fraction/mantissa
  - In some cases we need to truncate bits!
  - We may lose precision
  - If we only have three digits our example result becomes: 1.002



# Floating Point Things to Remember

---

- In decimal  $1/3$  is a repeating fraction  $0.3333 \dots$
- It will need to be cut off at some point
- If you quit at some fixed number of digits, then  $3 * 1/3 \neq 1$
- Floating Point arithmetic IS NOT associative
- $x + (y + z)$  is not necessarily equal to  $(x + y) + z$
- Addition may not even result in a change
- $(x + 1)$  MAYBE  $== x$
- Over and Underflows can occur
- Error Values
  - Overflow  $\rightarrow$  **inf/-inf** (means too large to be represented)
  - **Nan** for all other cases
- Huge impact on verifiability of programs!
- A limited number of integers can be represented exactly in floating point  $\rightarrow$  limited by length of fraction/mantissa
- $0.1 * 0.1 \neq 0.01$