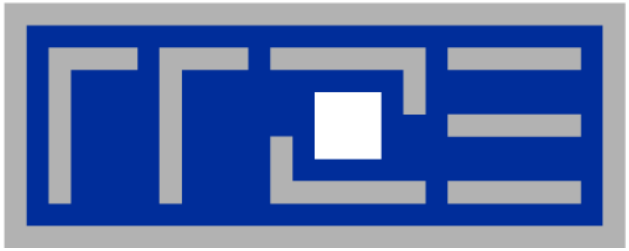


Programming Techniques for Supercomputers:
Introduction

Performance

Basics
Profiling
Measurement and Reporting

Benchmarks

Prof. Dr. G. Wellein[a,b]
Dr. G. Hager[a]
J. Hammer[b], C.L. Alappat[b]

[a]HPC Services – Regionales Rechenzentrum Erlangen
[b]Department für Informatik

University Erlangen-Nürnberg, Sommersemester 2019

Performance: Why thoroughly measure and report it?

- **Determine which computer is best suited for a given (set of) application(s)?**
  - Gaming PC or Atom based Laptop?
  - Cluster or fat server? Fast CPU? Intel or AMD or GPU???
  - Which applications? Which input/data sets?

- **Validate impact of new optimization / implementation / parallelization strategy and present to others**
  - Results need to be interpreted and potentially reproduced by external people
  - Compare with other / previous work
  - Justify efficient usage of expensive ressources

- **Determine capabilities for individual parts of the computer with (simple) kernels**
  - Data transfer / IO / computational capabilities
  - Often required to guide optimization strategies → Performance Modeling

Performance: What is a good measure/metric?

- **Performance = WORK / TIME**
- **"Pure" metrics – basic choices for "WORK"**
  - **MFlop/s: Millions of Floating Point Operations per Second**

$$\text{MFlop/s} = \frac{\text{Number of Floating Point Operations executed}}{10^6 * \textbf{TIME}}$$

(often cited for technical & scientific applications)

  - **MIPS: Millions of Instructions per Second**

$$\text{MIPS} = \frac{\text{Number of Instructions executed}}{10^6 * \textbf{TIME}}$$

(e.g. data bases, web servers ; computer architect view)

- **How to determine WORK, e.g. "Floating Point Operations"**
  - Count them manually (high level code / algorithm)
  - Use CPUs event counter → cf. LIKWID

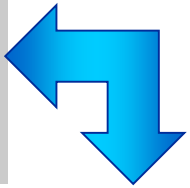
Performance – Pure WORK metric may fool…

- "My vector update code runs at 2,000 MFlop/s on a 2GHz processor!
- Great – isn't it?

```
for(i=0; i<n; i++)
{
    a[i]= 3.0*c0+c1*c2 +c3*c4*a[i] -1.d0 *a[i];
}
```

→ #FLOP = 8 * n

Same execution time but…



… but my MFlop/s rate is only ¼!

```
d0 = 3.0*c0+c1*c2;
d1 = c3*c4-1.d0;

for(i=0; i<n; i++)
{
    a[i]= d0 + d1*a[i];
}
```

→ #FLOP = 2* n + 5

→ Define WORK carefully – independent of implementation issues

Performance – choices for WORK

- **Iterations:** Total number of loop iterations performed: WORK = n iterations (see previous slide)
  → Performance metric**: Iterations / s**

- **Lattice Site/ Cell / Particle Updates:** Often used for stencil codes or Lattice Boltzmann fluid solvers: WORK = number of sites/cells/particles to be updated/computed
  → Performance metric**: Cell updates / s**

- **Physical simulation time:** Often used in molecular dynamics codes: WORK = Physical time (e.g. nanosenconds) a system is propagated
  → Performance metric: **nanoseconds / day**

- **Complete problem solution:** WORK: ”1” well defined problem
  → Performance metric**: 1 / s**

Performance – TIME

- Simplest performance metric (“Bestseller”): 1 / TIME
  - Measures time to solution
  - Carefully specify the “problem” you solved!
  - Best metric thinkable, but not intuitive in all situations (see later)

- Problem: Which TIME?

- LINUX / UNIX command `time` :

```
>time ./test.x
>34.650u 0.612s 0:35.28 99.9%

>time ./testwIO.x
>33.802u 0.608s 0:43.64 78.8%
```

- `> xxxu yyys  mm:ss  CPUratio%`

  `xxx` → USER CPU time [s] `yyy` → SYSTEM CPU time [s]
  `mm:ss` → Elapsed time `CPUratio` → (xxx+yyy)/mm:ss

- **Stay away from CPU time – it‘s evil!**

- **Elapsed time** (**walltime**) is the time you wait for your result!
  (Always use dedicated resource, e.g. one node)
- Measuring `walltime` within code on UNIX (-like) systems
  - Use `gettimeofday()` to measure timestamps:

```
#include <sys/time.h>

double timestamp(void){
   struct timeval tp;
   gettimeofday(&tp, NULL);
   return((double)(tp.tv_sec + tp.tv_usec/1000000.0)); }
```

  - WALL**TIME**:= Difference of two timestamps!
  - Code available in the exercise templates
  - Works fine for serial timings – due care for parallel apps is required

Where do I spend my time?

PROFILING

Performance: Where do I spend my time

- **How do I know where my code spends most of its time?**
- **This is called "Profiling"**
  - Application code is (automatically) instrumented such that runtime contributions of all subroutines, function, etc can be determined
  - Many more advanced profiling tools exist, e.g. Intel Amplifier, Oprofile, Codeanalyst – we start with simple one (gprof)
  - C++ code is notoriously hard to profile
    - Overloaded operators, tiny methods
  - Profiling may impact performance → Provides qualitative insight

Profiling with gprof

- Basic profiling tool under Linux: `gprof`
- Compiling for a profiling run (use compiler specific flag)

```
icc -pg …… -o a.out
./a.out
```

- After running the binary, a file `gmon.out` is written to current directory
- Human readable output via

```
gprof a.out
```

- Compiler inlining should be disabled for profiling
  - But then the executed code isn't what it should be…
- Profiling may (substantially) reduce overall code performance

Profiling with gprof: Example (sample - output)

```
tb082:/tmp> gprof ./lbmKernel-pg
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 80.05      3.17     3.17       10     0.32     0.32  relax_standard_flipped_il_2g_
 15.15      3.77     0.60        1     0.60     0.61  init_flipped_il_2g_
  3.79      3.92     0.15       10     0.01     0.01  bounceback_index_flipped_il_2g_
  0.51      3.94     0.02        2     0.01     0.01  make_bouncebacklist_
  0.25      3.95     0.01        1     0.01     0.01  obsin_
  0.25      3.96     0.01                             munmap
  0.00      3.96     0.00        2     0.00     0.00  get_time_info_
  0.00      3.96     0.00        1     0.00     3.95  MAIN__
  0.00      3.96     0.00        1     0.00     0.00  speed_info_mlups_

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

cumulative a running sum of the number of seconds accounted
 seconds   for by this function and those listed above it.

 self      the number of seconds accounted for by this
seconds    function alone.  This is the major sort for this
           listing.

calls      the number of times this function was invoked, if
           this function is profiled, else blank.

 self      the average number of milliseconds spent in this
ms/call    function per call, if this function is profiled,
           else blank.

 total     the average number of milliseconds spent in this
ms/call    function and its descendents per call, if this
           function is profiled, else blank.

name       the name of the function.  This is the minor sort
           for this listing. The index shows the location of
           the function in the gprof listing. If the index is
           in parenthesis it shows where it would appear in
           the gprof listing if it were to be printed.
```

Test of kernel routine:

- Initialize
- Run the 2 computational kernels 10 times

Profiling with gprof: Example (sample - output)

```
                     Call graph (explanation follows)


granularity: each sample hit covers 4 byte(s) for 0.25% of 3.96 seconds

index % time    self  children    called     name
                0.00    3.95       1/1           main [2]
[1]     99.7    0.00    3.95       1         MAIN__ [1]
                3.17    0.00      10/10          relax_standard_flipped_il_2g_ [3]
                0.60    0.01       1/1           init_flipped_il_2g_ [4]
                0.15    0.00      10/10          bounceback_index_flipped_il_2g_ [5]
                0.01    0.00       1/1           obsin_ [7]
                0.01    0.00       1/2           make_bouncebacklist_ [6]
                0.00    0.00       2/2           get_time_info_ [9]
                0.00    0.00       1/1           speed_info_mlups_ [10]
-----------------------------------------------
                                                  <spontaneous>
[2]     99.7    0.00    3.95                  main [2]
                0.00    3.95       1/1           MAIN__ [1]
-----------------------------------------------
                3.17    0.00      10/10          MAIN__ [1]
[3]     80.1    3.17    0.00      10         relax_standard_flipped_il_2g_ [3]
-----------------------------------------------
                0.60    0.01       1/1           MAIN__ [1]
[4]     15.4    0.60    0.01       1         init_flipped_il_2g_ [4]
                0.01    0.00       1/2           make_bouncebacklist_ [6]
-----------------------------------------------
                0.15    0.00      10/10          MAIN__ [1]
[5]      3.8    0.15    0.00      10         bounceback_index_flipped_il_2g_ [5]
-----------------------------------------------
                0.01    0.00       1/2           MAIN__ [1]
                0.01    0.00       1/2           init_flipped_il_2g_ [4]
[6]      0.5    0.02    0.00       2         make_bouncebacklist_ [6]
-----------------------------------------------
                0.01    0.00       1/1           MAIN__ [1]
[7]      0.3    0.01    0.00       1         obsin_ [7]
-----------------------------------------------
                                                  <spontaneous>
[8]      0.3    0.01    0.00                  munmap [8]
-----------------------------------------------
```

Butterfly graph

Who calls whom and how often?

Profiling with gprof: Example (C++)

- Example with wrapped `double` class:

```
class D {
  double d;
public:
  D(double _d=0) : d(_d) {}
  D operator+(const D& o) {
    D r;
    r.d = d+o.d;
    return r;
  }
  operator double() {
    return d;
  }
};
```

Main program:

```
const int n=10000000;
D a[n],b[n];
D sum;

for(int i=0; i<n; ++i)
  a[i] = b[i] = 1.5;

double s = timestamp();
for(int k=0; k<10; ++k) {
  for(int i=0; i<n; ++i)
    sum = sum + a[i] + b[i];
}
```

Profiling with gprof: Example (C++) profiler output

- `icpc -O3 -pg perf.cc`

```
  %   cumulative   self              self     total
 time   seconds   seconds    calls  Ts/call  Ts/call  name
101.01      0.41     0.41                             main
```

- `icpc -O3 -fno-inline -pg perf.cc`

```
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ns/call  ns/call  name
46.44      0.59     0.59 200000000     2.93     4.48  D::operator+(D const&)
29.63      0.96     0.37 240000001     1.56     1.56  D::D(double)
24.82      1.27     0.31                             main
```

- **But where did the time *actually* go?**
  - Butterfly (callgraph) profile also available
  - Real problem also with use of libraries (STL!)
  - Sometimes you have to roll your own little profiler

What does the hardware do?

PROBING HARDWARE PERFORMANCE

Probing Performance behavior

- **Once a hotspot is identified → determine the hardware utilization**

- **Performance counters allow to monitor processor events:**
  - The number and kind of instructions executed
  - The data transfers executed for each cache/memory level
  - The clock speed at which the processor runs
  - The power/energy consumption (starting with Intel Sandy Bridge architecture)
  - …
- **`likwid-perfctr` (from `likwid` toolbox) allows easy access to performance events and provides useful derived metrics, e.g. main memory bandwidth or Flop/s or cycles/instruction**

  http://code.google.com/p/likwid/

- **Separate lecture will cover that topic**

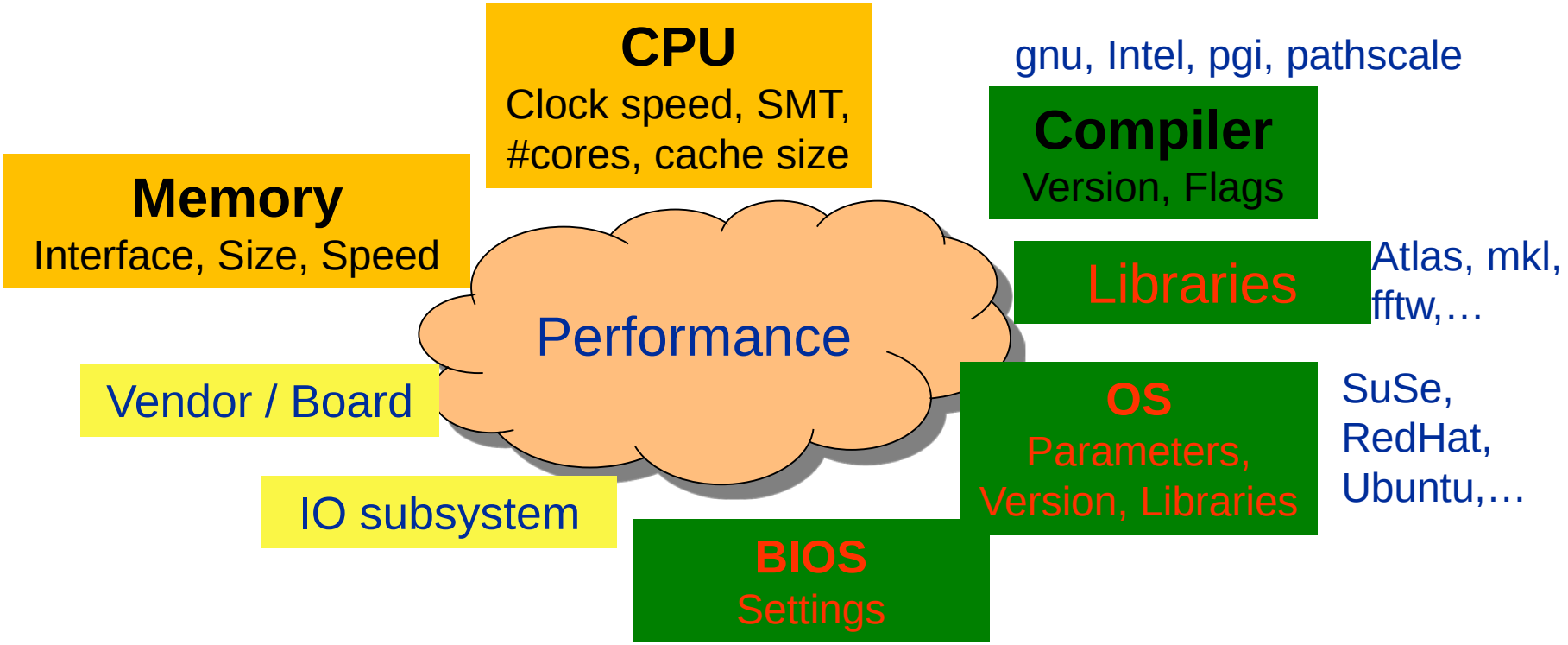
Measuring performance in a reproducible way

BEST PRACTICES FOR PERFORMANCE MEASUREMENT & REPORTING

“My code runs on an Intel Xeon Sandy Bridge processor 12 times faster than the results reported for code A in [xyz].”

Performance: Impact factors

- For a given code/problem performance may be influenced by many factors



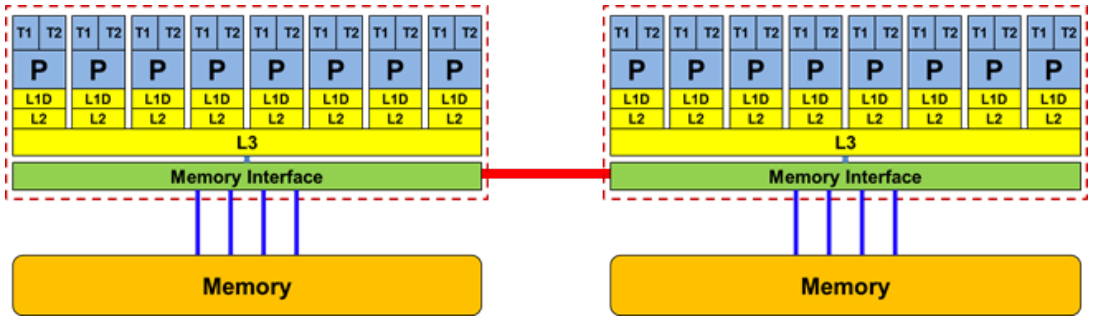
- For reproducibility of performance results all critical factors need to be reported!
- Sensibility and stability analysis!
- Statistics - fluctuations between runs

Important

Preparation

- Reliable timing/timer granularity (Minimum time which can be measured?)
- Document code generation (Flags, Compiler Version)
- Document system state (Clock, Turbo mode, Memory, Caches,…)
- Consider to automate runs with a skript (Shell, python, perl)

Doing



- Get **exclusive** system
- **Fix clock speed**
- Control **Affinity / Topology**– where does my code/threads/processes run exactly?
- Working set size – code input parameters?!
- Is result **deterministic** and **reproducible** →Statistics: Mean, Median, Best ??
- Basic **variations**: Thread count, affinity, working set size ←→ runtime
- Check: Are the **results reasonable**?

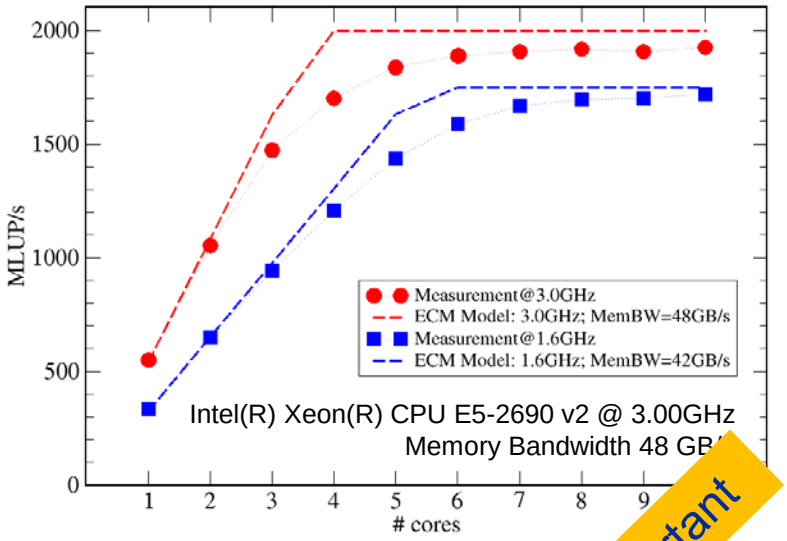
Important

Performance Measurement: Best Practices (cont.)

Postprocessing

- **Documentation**
- Plan **variations** to gain more information
- Many things can be better understood if you plot them (gnuplot, xmgrace)
- Use **statistics** to report performance fluctuations
- Try to understand and explain the result
- Is there a (simple) **model** which can (qualitatively) explain the performance levels and variations?

```
  do k = 1 , Nk; do j = 1, Nj
         do i = 1, Ni
                    y(i,j,k) = const*
  $                 ( x(i-1,j,k) + x(i+1,j,k)
  $                 + x(i,j-1,k) + x(i,j+1,k)
  $                 + x(i,j,k-1) + x(i,j,k+1) )
         enddo
enddo; enddo
```



Important

Benchmarks provide insights beyond the hardware fact sheet

BENCHMARKS

Benchmarks: Classification

1. **Real (full) applications:** Solves real world problem but includes everything and may run for hours or days on tens (or even more) processors! Portability problems
2. **Modified (restricted) applications**: Concentrate on most important part of the application, e.g. solver, and remove other parts (Pre-Postprocessing, IO). Reduce runtime to a reasonable amount (1,…,10 minutes), e.g. by performing a fixed number of iterative solver steps only.

3. **Kernels:** Extract small pieces, e.g. solver (→ LINPACK,…) or time consuming computational step (→ stream, (sparse) matrix vector multiplication, smoothing in MG,…). Easy to port, analyze and optimize. Evaluate individual features of a computer. Vendors report for several standard kernels (e.g. stream, LINPACK)

4. **Toy benchmarks:** Small pieces of code implementing popular algorithms (e.g. quicksort). Typical used for getting students started with programming.
5. **Synthetic benchmarks:** Simulate operations and data accesses of a variety of applications without having any relation to the application codes

Kernels are central for structured performance modelling!

Benchmarks – HPC standard benchmarks

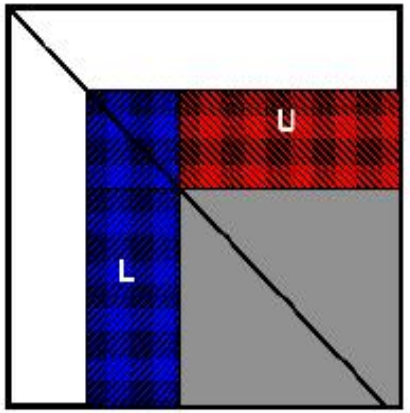
- **LINPACK → Top500 Ranking / Attainable peak performance**
- **STREAM → Attainable main memory bandwidth**
- **HPCCG → CG solver**
- SPEC → Industry standard – not HPC specific

Benchmarks – LINPACK: Towards Peak Performance

- Solve large dense linear system of equation, i.e.

$$A\,x = b$$

with A is a dense $(N \times N)$ matrix



- Algorithm: LU factorization of A
(+ forward/backward substitution) with
computational complexity $\frac{2}{3}\,N^3 + O(N^2)$

- Highly parallel implementations are available

- Achieves high fraction of machine peak performance (see 1st lecture)

(see http://www.netlib.org/benchmark/hpl/algorithm.html)

Benchmarks: HPCG – Something more realistic?

- HPCG: High Performance Conjugate Gradient benchmark
- Basic algorithm: Conjugate Gradient with a local symmetric Gauss-Seidel preconditioner
- Synthetic 3D sparse linear system (stencil-structure)
- Strong correlation with stream benchmark
- https://www.top500.org/hpcg/

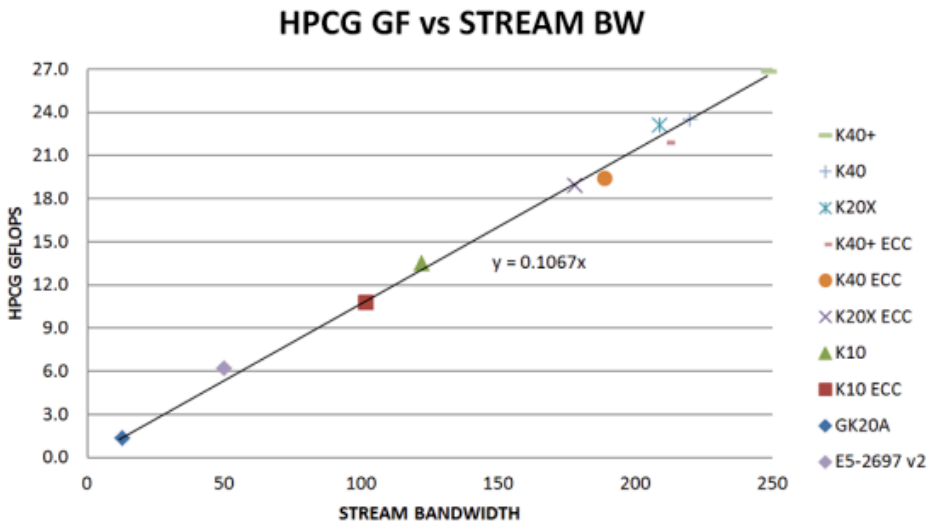


Figure from: https://devblogs.nvidia.com/parallelforall/optimizing-high-performance-conjugate-gradient-benchmark-gpus/

Benchmarks: STREAM – Capability of data paths

- http://www.cs.virginia.edu/stream/

- **Performs 4 “streaming” tests:**
  - COPY: A(1:N) = B(1:N)
  - Scale: A(1:N) = s*B(1:N)
  - Add: A(1:N) = B(1:N)+C(1:N)
  - Triad: A(1:N) = B(1:N)+s*C(1:N)

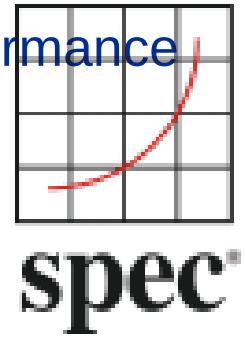
- **Results are reported in MB/s**

- **No changes are allowed**

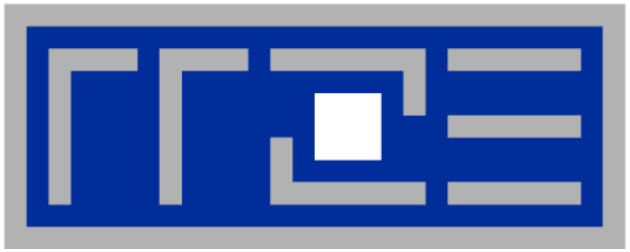
```
-------------------------------------------------
Double precision appears to have 16 digits of accuracy
Assuming 8 bytes per DOUBLE PRECISION word
-------------------------------------------------
-------------------------------------------------
STREAM Version $Revision: 5.6 $ common=ON
-------------------------------------------------
Array size =   33554432
Offset     =       1024
The total memory requirement is      768 MB
You are running each test  10 times
--
The *best* time for each test is used
*EXCLUDING* the first and last iterations
-------------------------------------------------
Number of Threads =           1
-------------------------------------------------
Printing one line per active thread....
-------------------------------------------------
Your clock granularity/precision appears to be      1 microseconds
-------------------------------------------------
Function     Rate (MB/s)  Avg time   Min time  Max time
Copy:      10758.0504      0.0499      0.0499      0.0499
Scale:     10380.3540      0.0517      0.0517      0.0518
Add:       11371.1566      0.0709      0.0708      0.0710
Triad:     11308.4169      0.0712      0.0712      0.0713
-------------------------------------------------
Solution Validates!
-------------------------------------------------
tb007:/tmp>
```

- **Stream & stream-like tests are used throughout the lecture**

Benchmarks: Modified/restricted applications (SPEC)



- Most widely accepted benchmark suites: SPEC (Standard Performance Evaluation Corporation) www.spec.org
- Long history: Since 1980's
- Several categories of SPEC benchmarks:
  - CPU
  - Graphics/Workstations
  - MPI / OpenMP
  - Java Client/Server
  - Mail Server
  - …
- Benchmarks are reported in wallclocktime and compared with a reference system
- Reference: Sun UltraSparc II@296MHz (1997): ~5 Million transistors (cf. Nehalem EX: 2.3 Billion, nVIDIA FERMI: 3 Billion)



Some DON'Ts for presenting performance results

"Fooling the masses with performance results: Old classics and some new ideas"

45 – 60 minute presentation by G. Wellein & G. Hager motivated by
D. Bailey, "*Twelve Ways to Fool the Masses When Giving Performance Results on Parallel Computers*"(1991)

(see moodle)

See also:
http://blogs.fau.de/hager/archives/category/fooling-the-masses