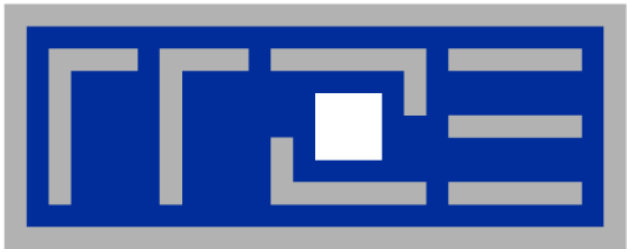


Unveiling some Mysteries of Application Performance on Multi-/Manycore Processors

Gerhard Wellein, Georg Hager
Erlangen Regional Computing Center & Dept. of Computer Science
Friedrich-Alexander-University Erlangen-Nuremberg

As time goes by….

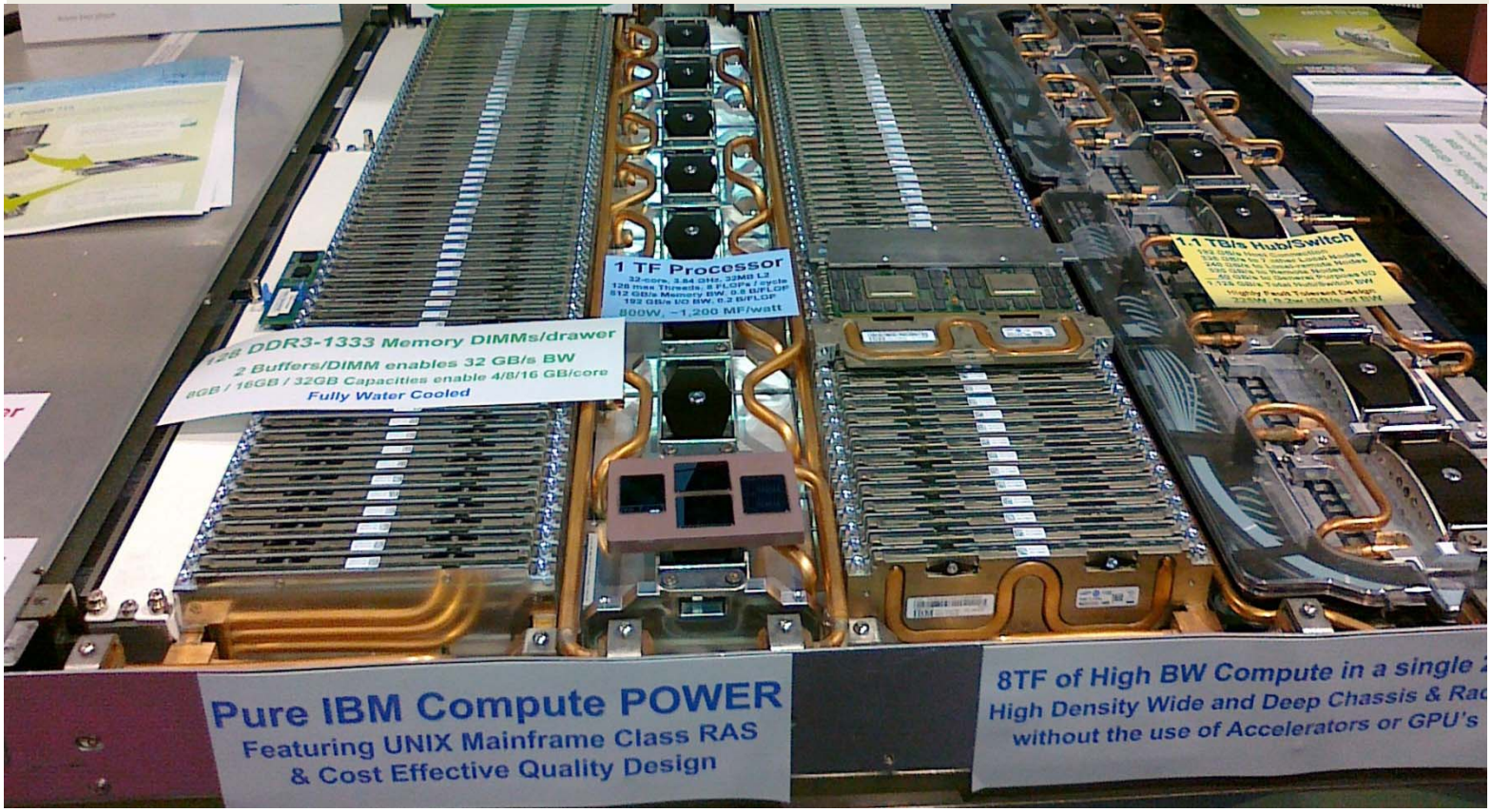
MediaMarkt 2003

MediaMarkt 2011

The end of the clock speed race

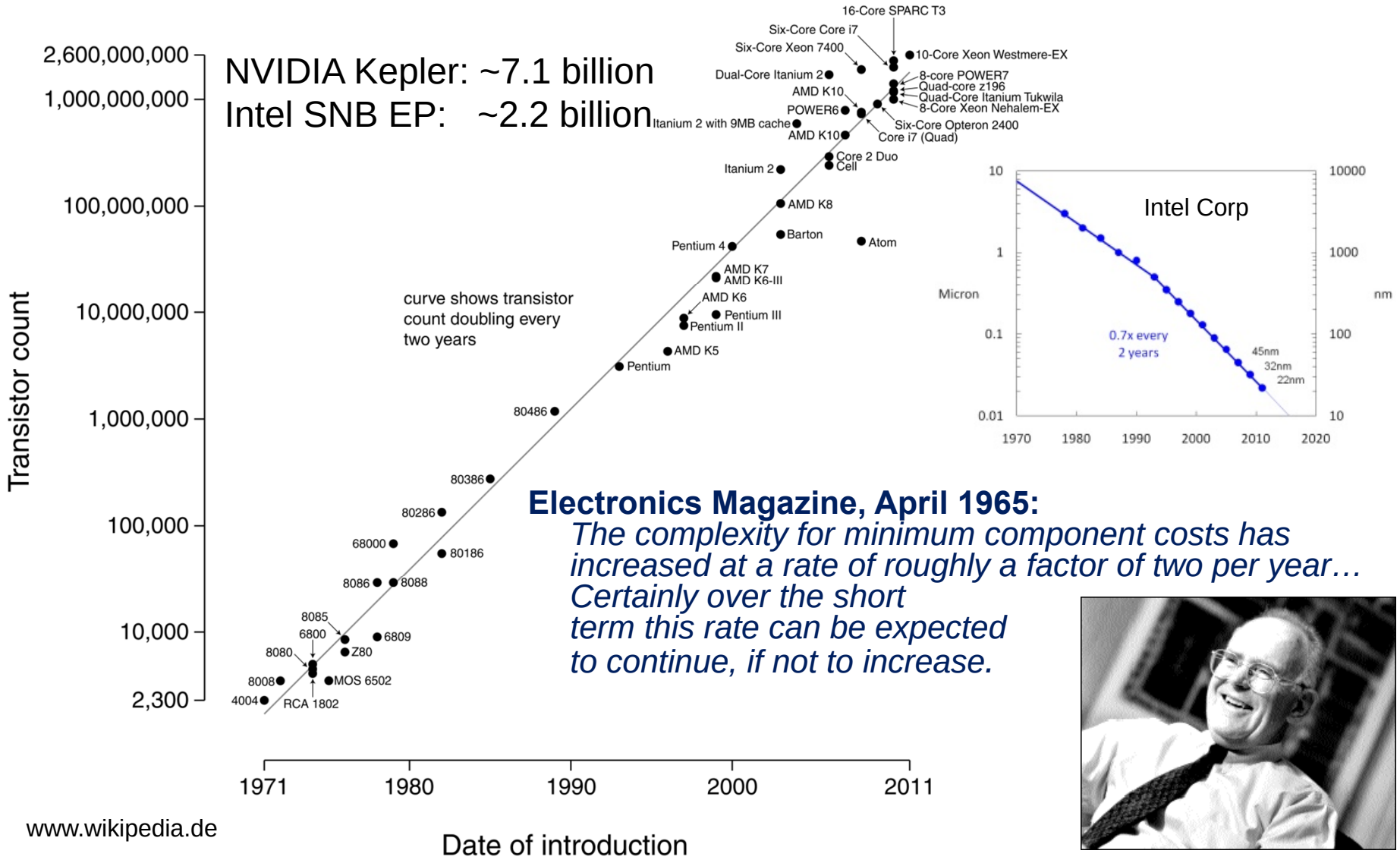
Exponential growth of x86-CPU clock speed for 15+ years
Since 2004 the 4 GHz barrier limits x86 clock speed

Higher clock speeds require special efforts



- **Technology trends and state-of-the-art multi-/manycore processors**
- ***Mysteries***
  - The programming language is critical for performance
  - Even with simple loop structures the compiler fails to parallelize / vectorize
  - Performance is a black box – take what you get
  - Sometimes it is so easy to scale on multicore chips
  - Erratic performance numbers on multi-processor nodes (NUMA)

Technology trend: Moore's law continues…



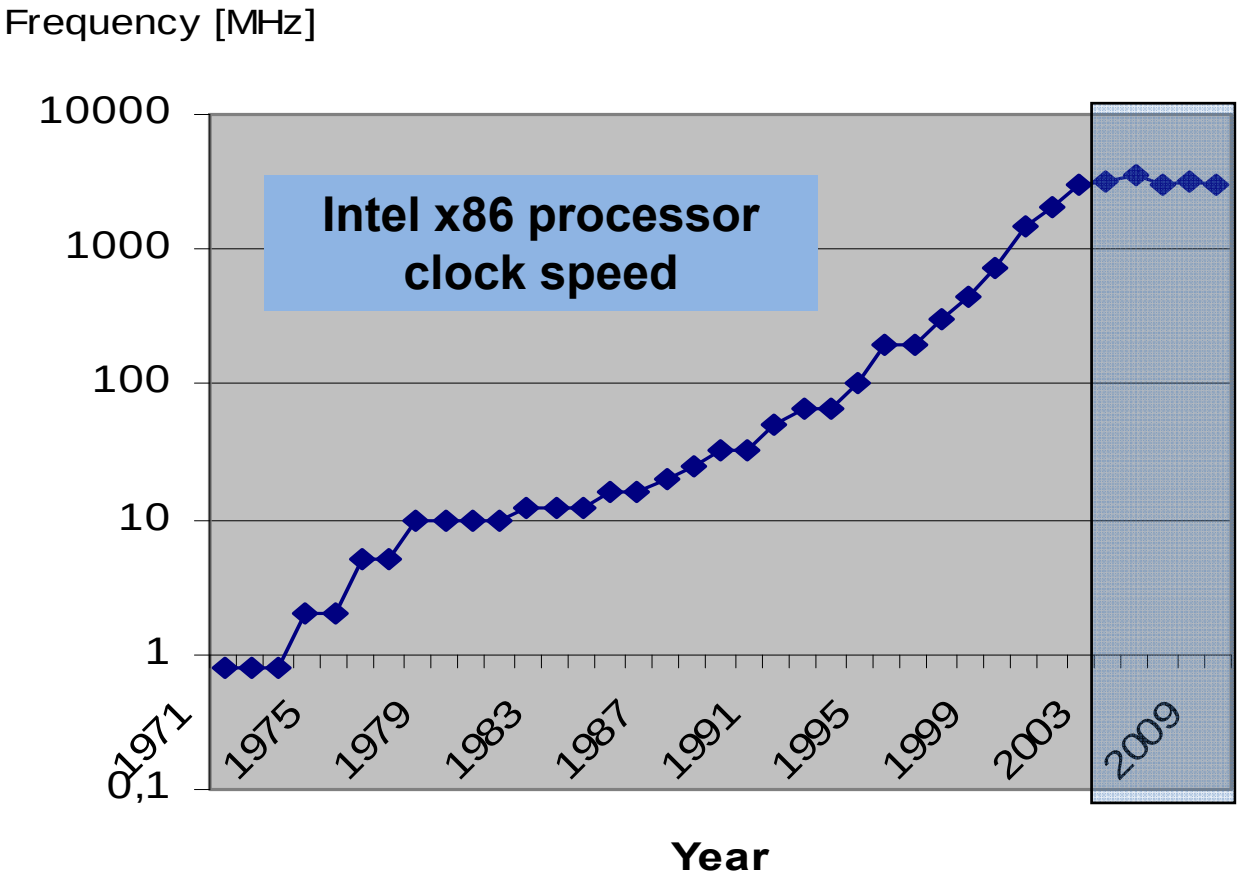
www.wikipedia.de

Electronics Magazine, April 1965:

The complexity for minimum component costs has increased at a rate of roughly a factor of two per year… Certainly over the short term this rate can be expected to continue, if not to increase.

Technology trend: … but the free lunch is over

- **Moore's law** → **run smaller transistors faster**
- **Faster clock speed** → **Higher Throughput (Ops/s) for free**



Single core: Instruction level parallelism:

- Superscalarity
- **Single Instruction Multiple Data (SIMD) → SSE / AVX**

Investing the transistor budget:

- **Multi-Core/Threading**
- **Complex on chip caches**
- New on-chip functionalities (GPU, PCIe,…)

The physical constraint: Power consumption

- Power consumption (P_T) per transistor: $P_T \sim V_C^2 * f$
- Supply voltage approaches a lower limit: $V_C \sim 1\ V$
- Power consumption / chip: $P \sim \#Transistors * P_T$
- Max. P approaches economical limit: $P_{max} \sim 80\ W,\ldots,130\ W$

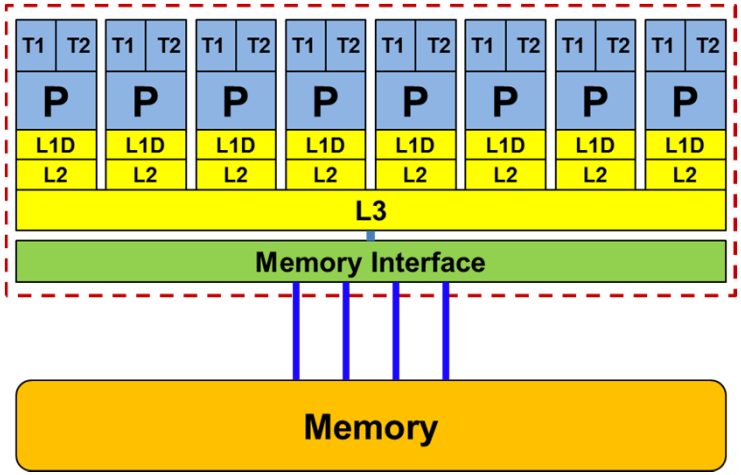
P5 / 80586 (1993)	Pentium3 (1999)	Pentium4 (2003)	Core i7–3960X (2012)
66 MHz	600 MHz	2800 MHz	3300 MHz
16 W @ V_C = 5 V	23 W @ V_C = 2 V	68 W @ V_C = 1.5 V	130 W @ V_C = 1.3
800 nm / 3 M	250 nm / 28 M	130 nm / 55 M	32 nm / 2200 M
			Hexa-Core

P_{max} / Core supply voltage

Structure Size / #Transistors

Be prepared for more cores with less complexity and slower clock!

There is no single driving force for chip performance!



Intel Xeon
"Sandy Bridge EP" socket
4,6,8 core variants available

Floating Point (FP) Performance:

$$P = n_{core} * F * S * \nu$$

n_{core}	number of cores:	8
F	FP instructions per cycle: (1 MULT and 1 ADD)	2
S	FP ops / instruction:	4 (dp) / 8 (sp)
	(256 Bit SIMD registers – "AVX")	
ν	Clock speed :	~2.7 GHz

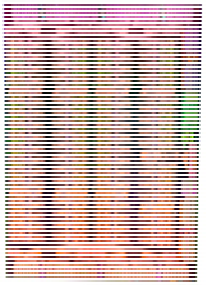
TOP500 rank 1 (1995)

P = 173 GF/s (dp) / 346 GF/s (sp)

But: P=5 GF/s (dp) for serial, non-SIMD code

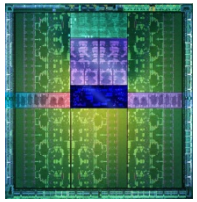
Highly parallel on-chip architectures: Accelerators

- **Intel Xeon/Phi**
  - **60+ IA32 cores** each with 512 Bit SIMD FMA unit → **960 "SIMD SP tracks"**
  - **Clock Speed: ~1000 MHz**
  - Transistor count: ~3 B (22nm)
  - Power consumption: ~250 W



  - Peak Performance (DP): ~ 1 TF/s
  - Memory BW: ~250 GB/s (GDDR5)
  - Threads to execute: 100-200
  - Programming: Fortan/C/C++ +OpenMP + vectorization

- NVIDIA Kepler (GK110)
  - 15 SMX units each with 192 "SP cores" → **2880 "SP cores"** in total
  - **Clock Speed: ~700 MHz**
  - Transistor count: 7.1 B (28nm)
  - Power consumption: ~250 W



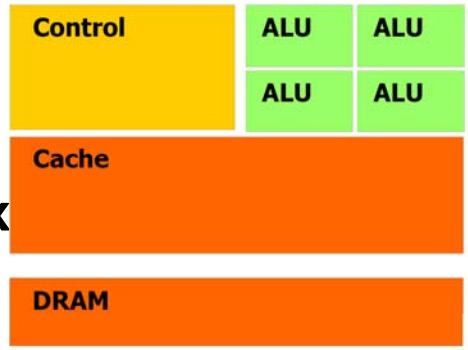
  - Peak Performance (DP): ~ 1 TF/s
  - Memory BW: ~ 250 GB/s (GDDR5)
  - Threads to execute: 10.000+
  - Programming: CUDA, OpenCL, (OpenACC)

TOP7: "Stampede" at Texas Center for Advanced Computing	**TOP500 rankings**	TOP1: "Titan" at Oak Ridge National Laboratory

Trading single thread performance for parallelism: GPGPUs vs. CPUs

GPU vs. CPU light speed estimate:

1. **Compute bound (DP): 2x-4x**
   Compute bound (SP): 5-10x
2. **Memory Bandwidth: 2x-4x**



	Intel Core i7 –3960x ("Sandy Bridge")	Intel Xeon E5-2680 DP node ("Sandy Bridge")	NVIDIA K20 ("Kepler")
Cores@Clock	6 @ 3.3 GHz	2 x 8 @ 2.7 GHz	2496 @ 0.7 GHz
Performance⁺/core	52.8 GFlop/s	43.2 GFlop/s	1.4 GFlop/s
Threads@STREAM	<6	<16	>8000?
Total performance⁺	315 GFlop/s	691 GFlop/s	3,500 GFlop/s
Stream BW	~40 GB/s	2 x ~40 GB/s	168 GB/s (ECC=1)
Transistors / TDP	2.2 Billion* / 130 W	2 x (2.27 Billion/130W)	**7.1 Billion/250W**

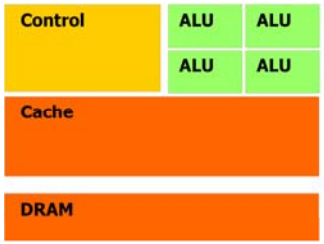
⁺ ***Single Precision***

** Includes on-chip GPU and PCI-Express*

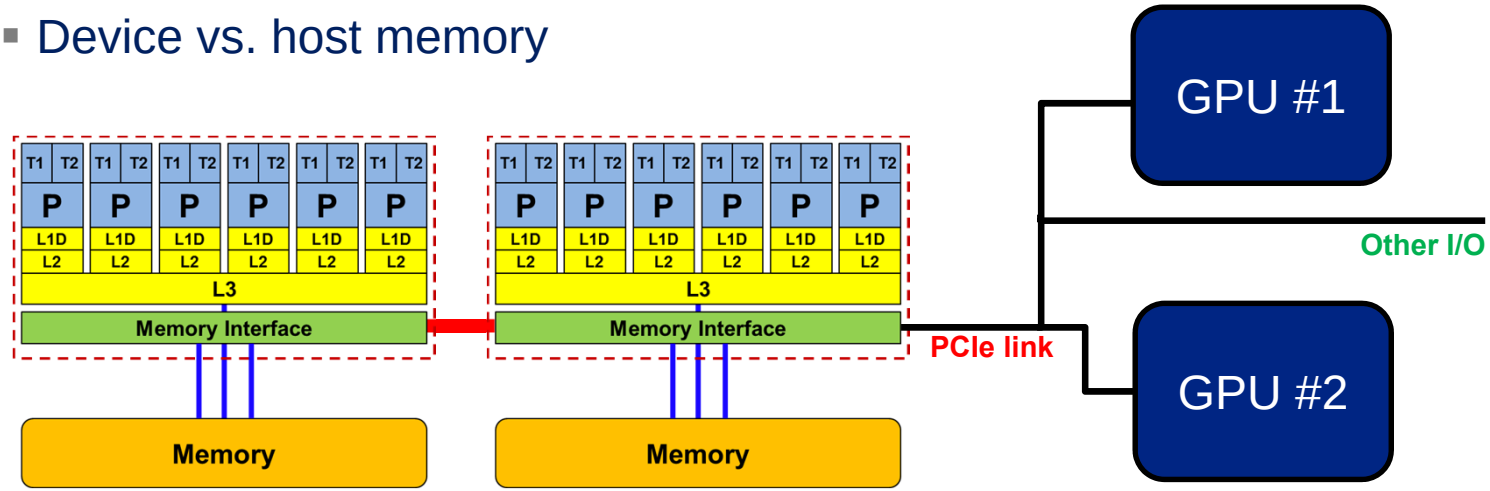
Complete compute device

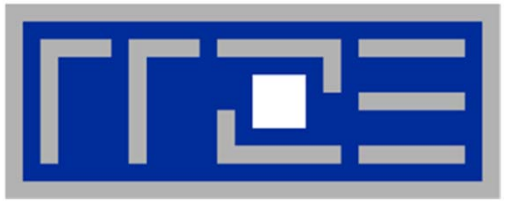
Trends to remember

- **“Few” cores@high clock speeds**
  ←→
  Massive number of
  execution units@low clock speed



- **Complex topology issues within compute node**
  - Simultaneous Multi-Threading
  - Shared vs. dedicated caches
  - ccNUMA
  - Device vs. host memory





Mystery

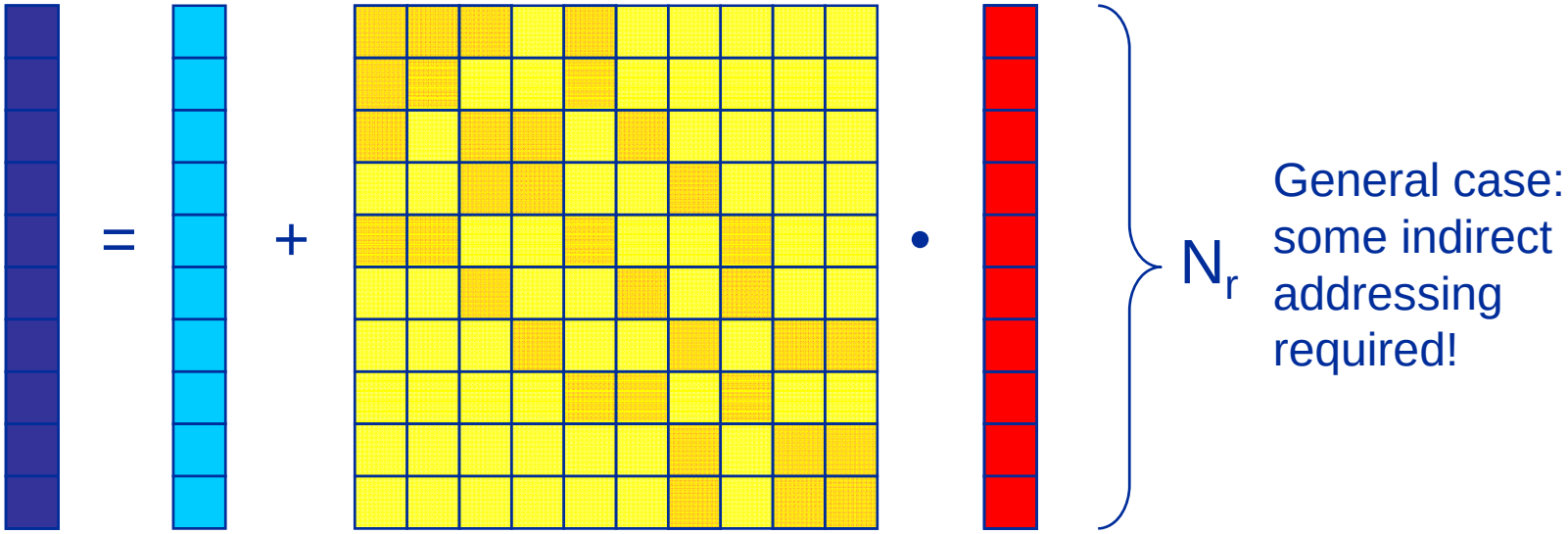
The programming language is critical for performance

C / C++ / FORTRAN,…, Java or OpenCL
Application scenario: sparse matrix-vector multiply

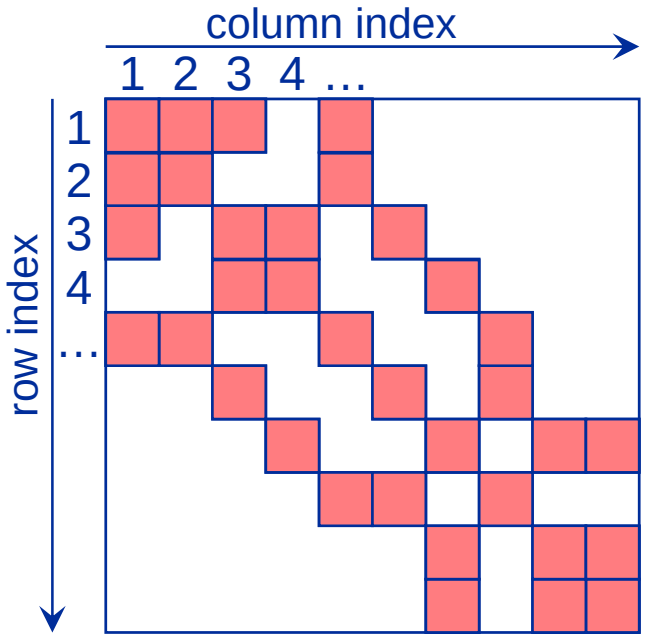
Sparse matrix-vector multiply (spMVM)

- **Key ingredient in sparse solvers for Finite-Element-Method or in Quantum Physics/Chemistry**

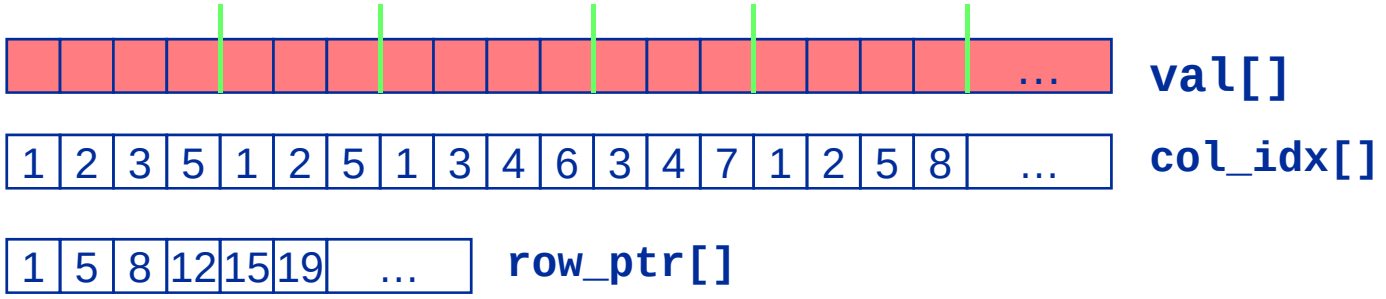
- **Store only N_{nz} nonzero elements of matrix and RHS, LHS vectors with N_r (number of matrix rows) entries**
- **"Sparse": $N_{nz} \sim N_r$**



CRS matrix storage scheme



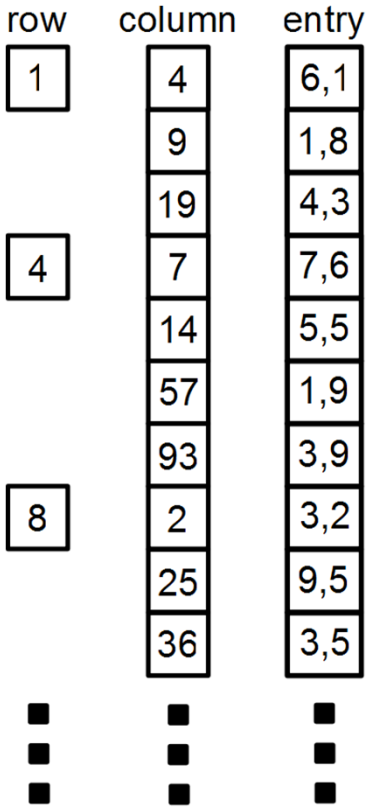
- **`val[]`** stores all the nonzeros (length N_{nz})
- **`col_idx[]`** stores the column index of each nonzero (length N_{nz})
- **`row_ptr[]`** stores the starting index of each new row in **`val[]`** (length: N_r)



Impact of programming language/style
A representative example ?!

- **Sparse matrix-vector multiplication: `y = y + M*x`**

- **The classical Fortran/C approach:**
  Compressed Row Storage (CRS) for matrix `M`



Sparse MVM code snippet:

```
for(i = 0; i< number_of_unknowns; ++i){
    for(j = row(i); i < row(i+1);++j){
      y[i] =y[i] +entry[j] *x[column[j]];
}}}
```

Impact of programming language/style
A representative example?!

- spMVM (within Finite-Element Method): `y = y + M*x`

- The object oriented C++ approach for FEM:
  Consider each row of M as an object, e.g. the stencil of a node in FEM → Matrix `M` is a vector of "stencils"

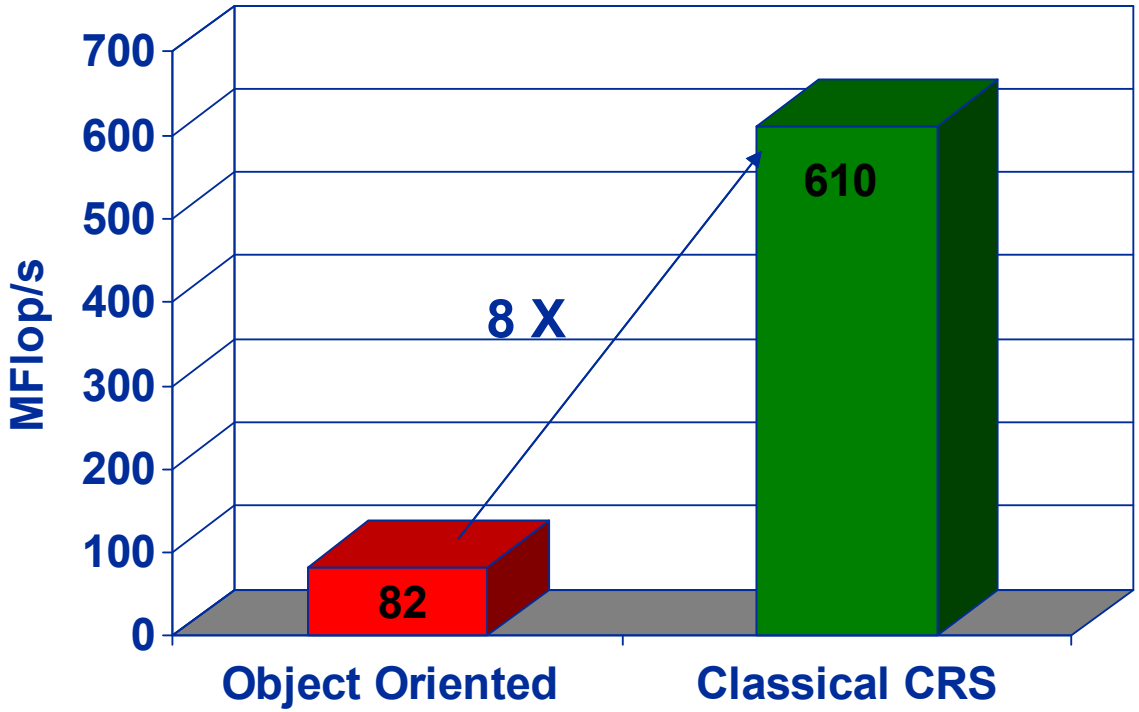
```
for(i=0; i < number_of_unknowns; ++i){
   for(j=0; j < stencil_array[i].m_row_stencil_length;++j){

      y.m_vektor[i] = y.m_vektor[i] +
            stencil_array[i].m_row_stencil[j] *
            x.m_vektor[stencil_array[i].m_row_position[j]];
}}}
```

```
//Class Stencil
class Stencil{ int m_row_stencil_length; double *m_row_stencil;
int *m_row_position; }
```

Impact of programming language/style
A representative example?!

- **FEM-oriented spMVM: `y = y + M*x`**
- **Problem: FEM on semi structured grid with 55056 vertices**
- **Serial Performance of simple CG solver including spMVM**
- **Testmachine:**
  Intel Core2 2 .26GHz (Penryn)

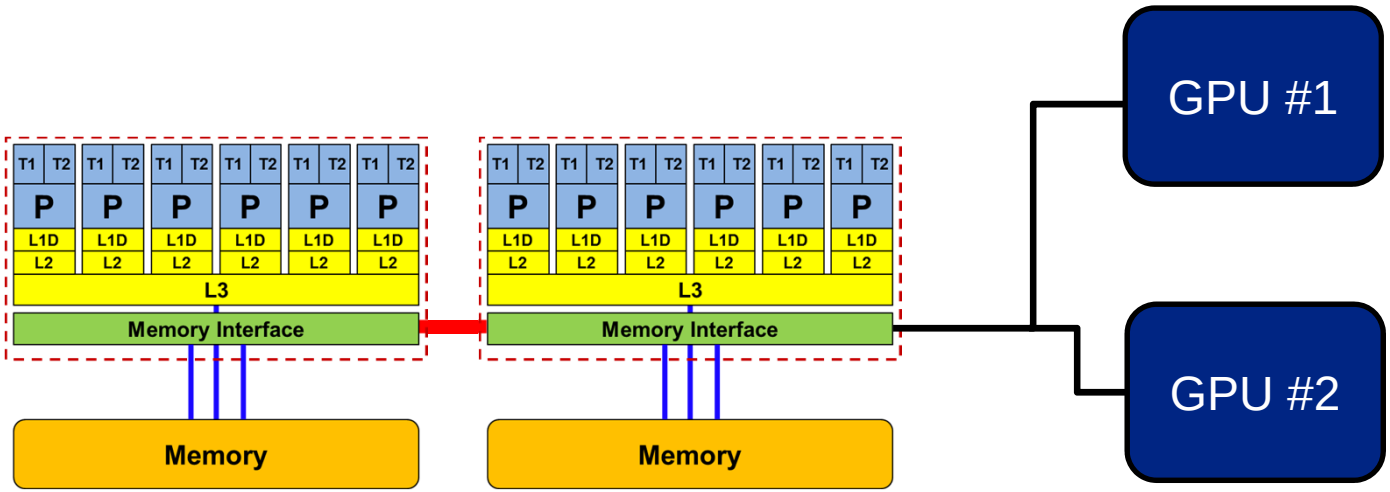


→ Object orientation may be orthognal to performance

Programming for heterogeneous systems: A unified code for CPU and Accelerators?

```
size_t i = get_global_id(0);
if (i < number_of_unknowns) {
   for(int j=row[i]; j<row[i+1]; ++j) {
      y[i] = y[i] + entry[j]*= x[column[j]];}}
```

OpenCL

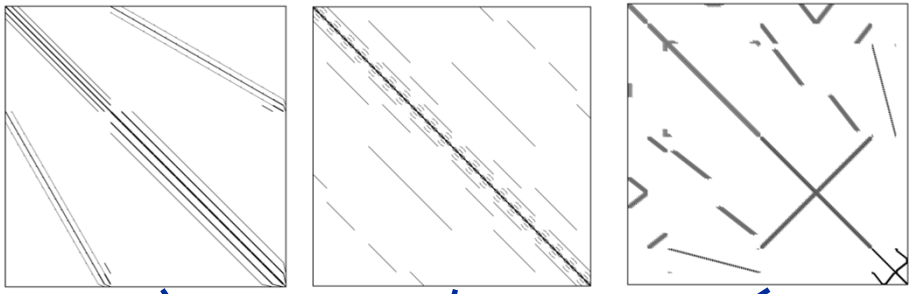


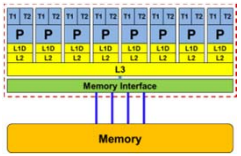

```
#pragma acc for
for(i = 0; i< number_of_unknowns; ++i){
    for(j = row(i); i < row(i+1);++j){
      y[i] =y[i] +entry[j] *x[column[j]];}}
```

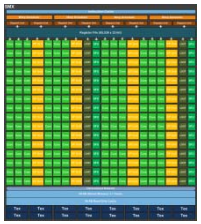
OpenACC
DIRECTIVES FOR ACCELERATORS

Programming for heterogeneous systems: A unified code for CPU and Accelerators?

- All kernels written in OpenCL
- Data format is the key to performance!
- Even with OpenCL: CPU and GPU code branch







	spMVM format	dlr1	rrze3	RM07R	copy BW	GPU/ CPU
Intel Xeon E5-2690	CRS	6.5	4.5	6.2	39 GB/s	1
Tesla K20c (*Kepler*)	CRS	1.3	1.6	1.8	144 GB/s	3.7
	ELL-R	22.5	14.0	13.0		
	Best	22.6	15.7	19.9		

- Best data layout: 1 Kepler / 1 Intel Xeon processor: ~3.2-3.4 (As suggested by STREAM bandwidth)

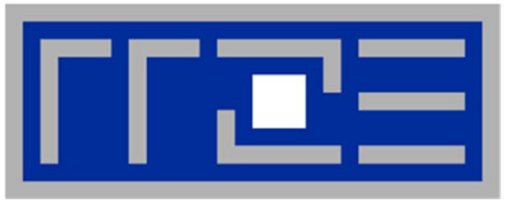
A few comments on programming language/style

- **Don’t be religious about programming style!**
- **IF you program C++ like Fortran → Fortran performance**
- **BUT do students still know Fortran?!**

- **Adopt programming style to problem and hardware (if performance is critical for you)**
- **Performance for a complex problem on a complex hardware is NOT for free**

- **Conservation law of hardware efficient programming**

*PERFORMANCE * FLEXIBILITY = constant*



Mystery

Even with simple loop structures
the compiler fails to parallelize / vectorize

Intel compiler
Prototype scenario: 3D stencil / Jacobi

Compiler based parallelization for 3D stencil

- **“Jacobi iteration”**
  - Finite difference discretization of Laplace equation in 3D
  - Use Jacobi method to solve the corresponding linear system of equations
  - Prototype for many regular stencil update scheme, e.g. in Multigrid schmes

```
void jacobi_full( double *Y, const double *X, int size) {
  int i,j,k,ofs;
#pragma omp parallel for private(ofs,i,j,k)
    for(i=1; i<size-1; ++i) {
      for(j=1; j<size-1; ++j) {
       ofs = i*size*size + j*size;
       for(k=1; k<size-1; ++k) {
         Y[ofs+k] = oos*(X[ofs+k+1]+X[ofs+k-1]+X[ofs+k-size]+
           X[ofs+k+size]+X[ofs+k-size*size]+X[ofs+k+size*size]);

       }}}}
```

- **No data dependencies → easy to parallelize**
- **Is the compiler clever enough?!**

Compiler based automatic parallelization

- **Compiler: Intel C-compiler (Version 13.0.1.117 Build 20121010)**
- **1st try:**

```
>icc -O3 -xHOST -par-report2  -parallel -c j3d_c.c
j3d_c.c(34): (col. 5) remark: loop was not parallelized: existence of parallel dependence.
j3d_c.c(35): (col. 7) remark: loop was not parallelized: existence of parallel dependence.
j3d_c.c(37): (col. 2) remark: loop was not parallelized: existence of parallel dependence.
```

- **Pointer aliasing in C → Compiler assumes pot. dependency: Y←→X**
- **2nd try:**

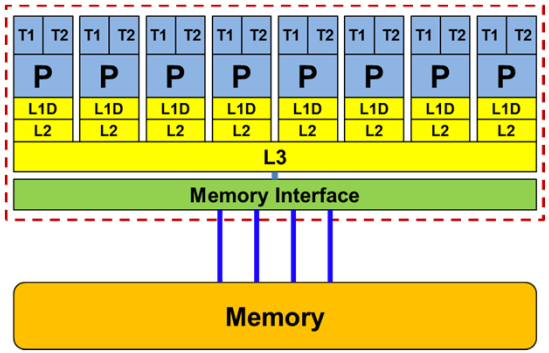
```
>icc -O3 -xHOST -par-report2  -parallel –fno-alias -c j3d_c.c
…
j3d_c.c(34): (col. 5) remark: loop was not parallelized: existence of parallel dependence.
j3d_c.c(35): (col. 7) remark: loop was not parallelized: existence of parallel dependence.
j3d_c.c(37): (col. 2) remark: loop was not parallelized: insufficient computational work.
```

- **3rd try:**

```
>icc -O3 -xHOST -par-report2  -parallel –fno-alias –par-threshold0 -c j3d_c.c
…
j3d_c.c(37): (col. 7) remark: LOOP WAS AUTO-PARALLELIZED.
```

Compiler based (automatic) parallelization

- Intel Xeon E5-2690 (8 cores, 2.9 GHz)
- Testcase: 240^3
- Serial performance: 521 MLUP/s (=3.1 GF/s)



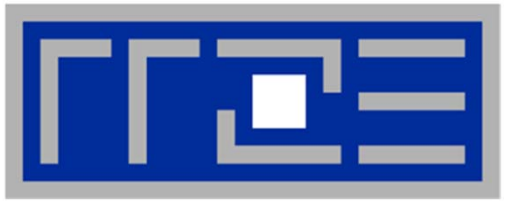
- Parallel performance (higher is better)

threads	Compiler	OpenMP
1	312	519
2	195	902
4	181	1353
6	156	1390
8	151	1377

- Compiler version suffers from overhead of inner loop parallelization!

Support our compiler!

- **Recognizing data parallel structures is essential for automatic parallelization / vectorization**
- **Compiler has a limited view of the code**
- **Do not completely rely on the compiler (or other software layer) support it:**
  - Plain programming – do not hide information!
  - Use compiler directives / pragmas / …. to support vectorization and parallelization
  - Inlining often helps
  - Even with highest optimization level your code should produce correct results

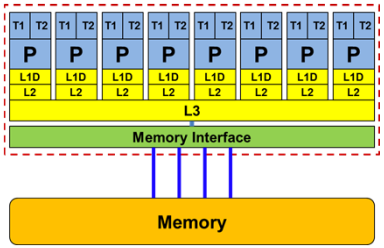


Mystery

Performance is a black box – take what you get

Prototype scenario: Jacobi

Performance is a black box



- **Equivalent Fortran version**

```
!$OMP PARALLEL DO
do k = 1 , N
  do j = 1 , N
   do i = 1 , N
     y(i,j,k)= b*(  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
```

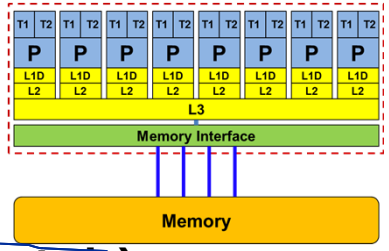
- **BTW:**
  - Compiler parallelizes outer loop in Fortran
  - Performance of Fortran is the same as OpenMP C-Code (assuming comparable compiler switches)

- **What is the maximum performance on Intel Xeon E5-2690 (8 cores, 2.9 GHz)?**

Performance is a black box

```
!$OMP PARALLEL DO
do k = 1 , N
  do j = 1 , N
   do i = 1 , N
     y(i,j,k)= b*( 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
```

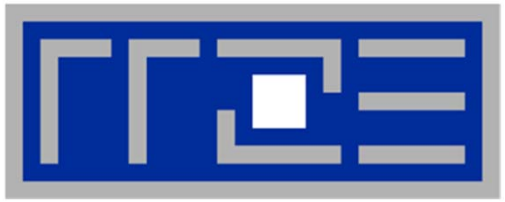
Requires "Write Allocate"



Elements have been loaded in previous (i,j,k) iterations

- **Assumption: Main memory bandwidth limits Jacobi performance**
- **STREAM bandwidth~ 36 GB/s**
- **How many data must be transferred between processor and main memory for a single Lattice Update (assume DP)**
  - `y(i,j,k)`: (**1 STORE** + **1 LOAD**) * 8 B → **16 B/LUP**
  - `x(i,j,k+1)` **1 LOAD** * 8B → **8 B/LUP**

→ Maximum Performance: (36 GB/s) / (24 B/LUP) = 1500 MLUP/s



Mystery

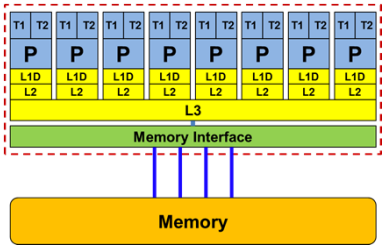
Sometimes it is so easy to scale on multicore chips

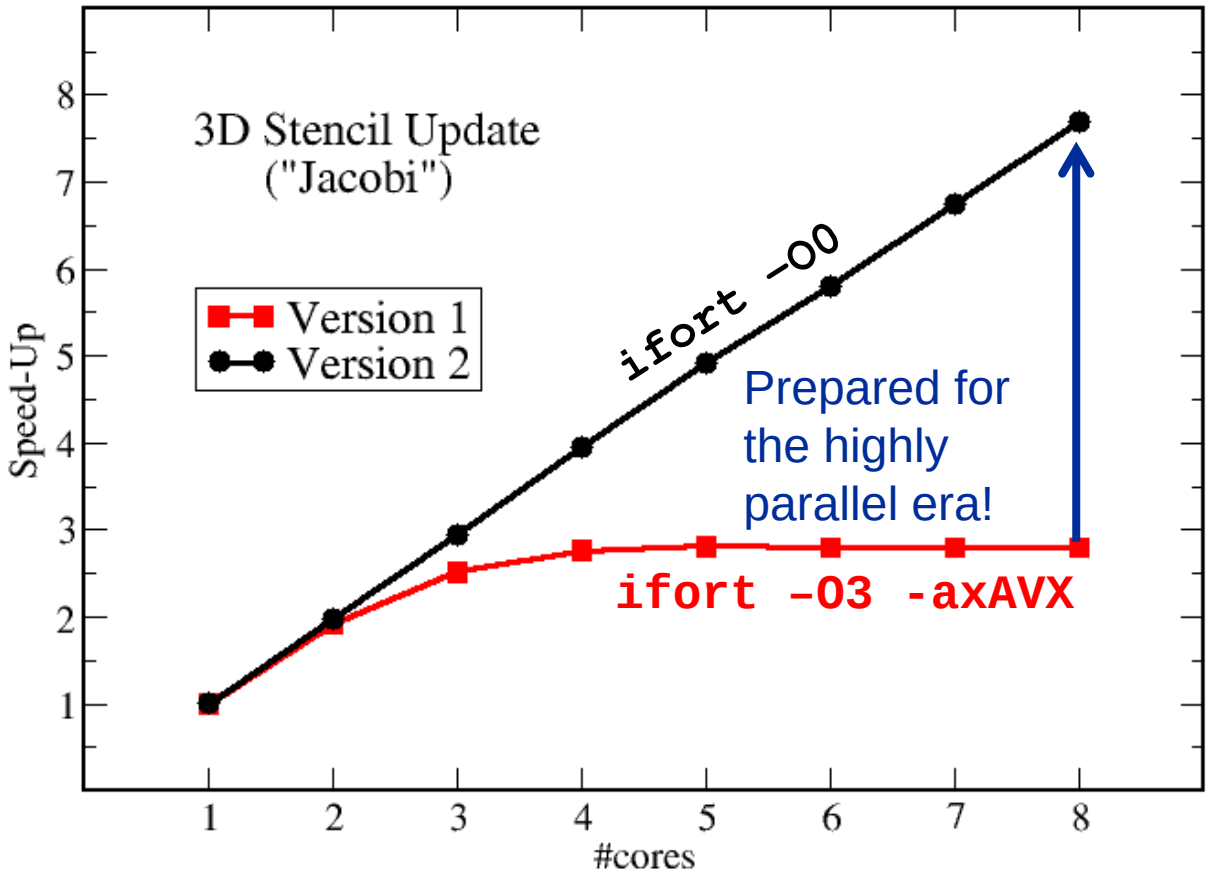
Prototype scenario: Jacobi

Multicore scalability mystery: Jacobi iteration

```
!$OMP PARALLEL DO
do k = 1 , N
  do j = 1 , N; do i = 1 , N
     y(i,j,k)= b*(  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
```

Changing only a few letters makes this code scalable on a 8-core chip





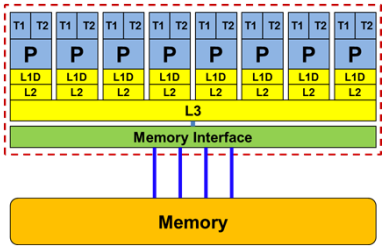
Multicore scalability mystery: Jacobi iteration

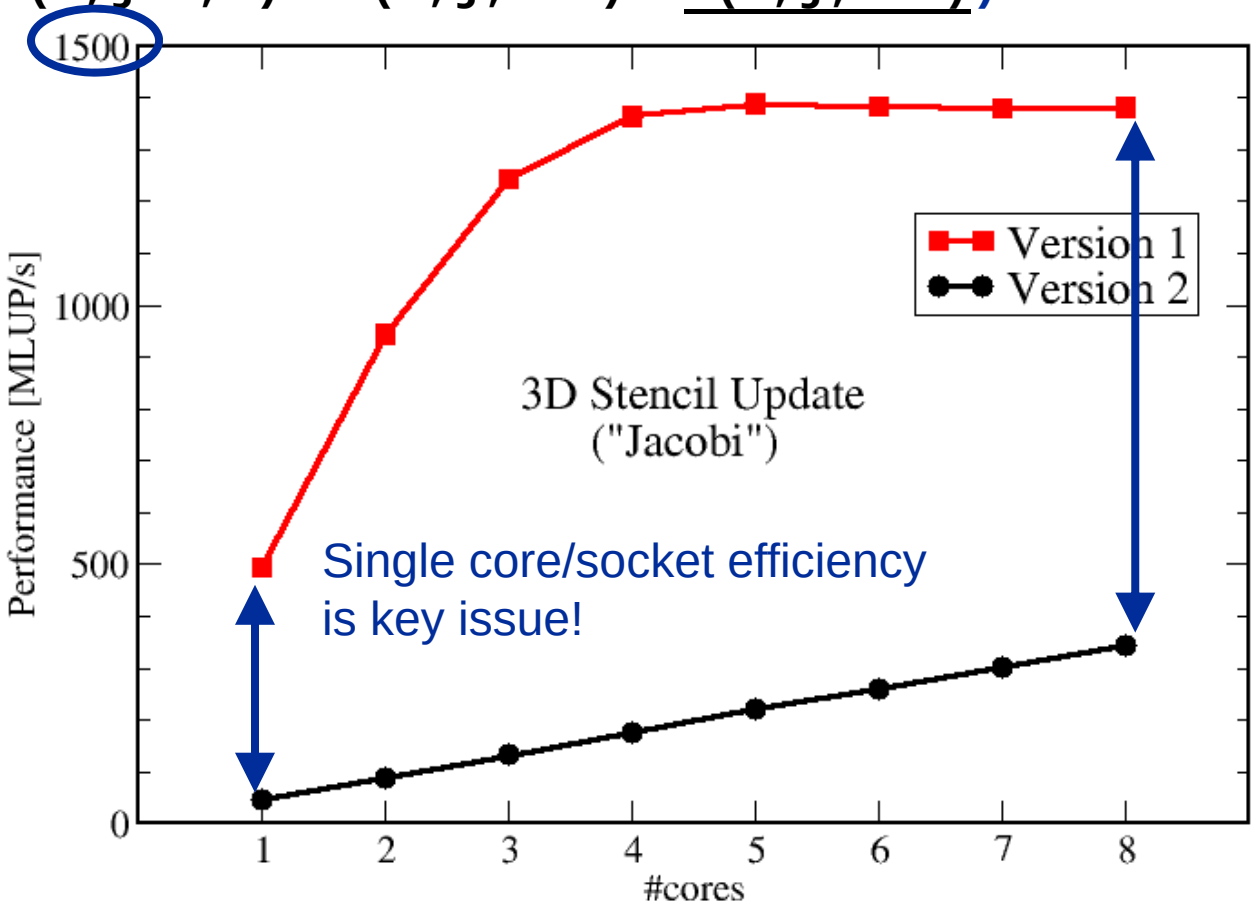
```
!$OMP PARALLEL DO
do k = 1 , Nk
  do j = 1 , Nj; do i = 1 , Ni
     y(i,j,k)= b*(  x(i-1,j,k)+ x(i+1,j,k)+ x(i,j-1,k)+
                    x(i,j+1,k)+ x(i,j,k-1)+ x(i,j,k+1))
    enddo; enddo
enddo
```

Simple performance model:

- 24 Byte / LUP
- Socket Bandwidth: 36 GB/s

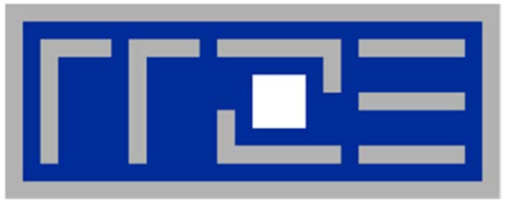
→ Max. 1500 MLUP/s





Summary: Performance and Scalability

- When optimizing / parallelizing performance critical code (even simple) performance models help a lot
- Having a good estimate of the optimal runtime of a code is the first step of any optimization/parallelization attempt
- Achieving scalability is easy → Compare with a bad baseline ("Slow computing")
- Single core/thread/process performance should be the first target



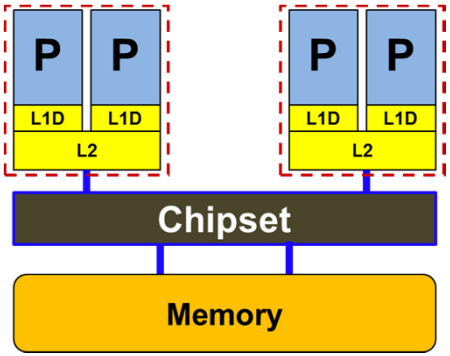
Mystery

Erratic performance numbers on multi-processor nodes (ccNUMA)

Prototype scenario: STREAM

From UMA to ccNUMA
Basic architecture of commodity compute cluster nodes

Yesterday (2006): Dual-socket Intel "Core2" node:

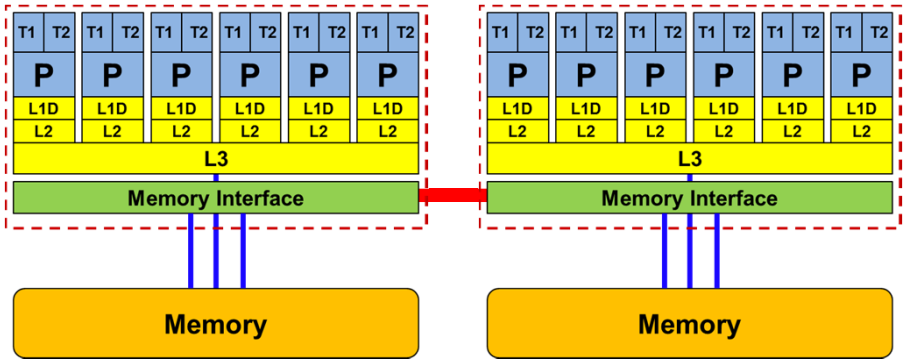


Uniform Memory Architecture (UMA)

Flat memory ; symmetric MPs

But: system "anisotropy"

Today: Dual-socket Intel (Westmere) node:



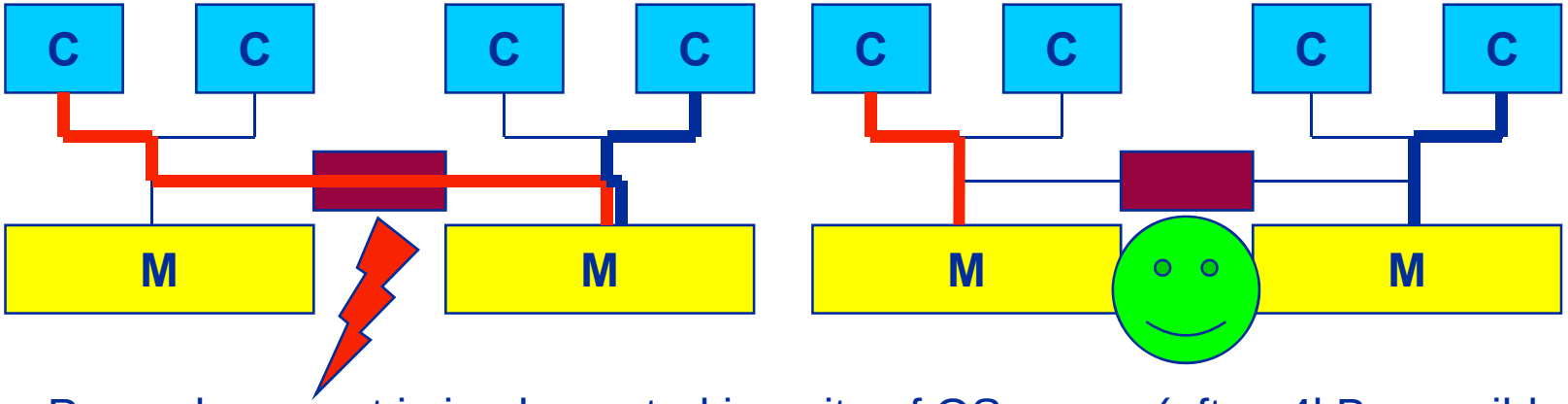
Cache-coherent Non-Uniform Memory Architecture (ccNUMA)

HT / QPI provide scalable bandwidth at the price of ccNUMA architectures: *Where does my data finally end up?*

On AMD it is even more complicated → ccNUMA within a socket!

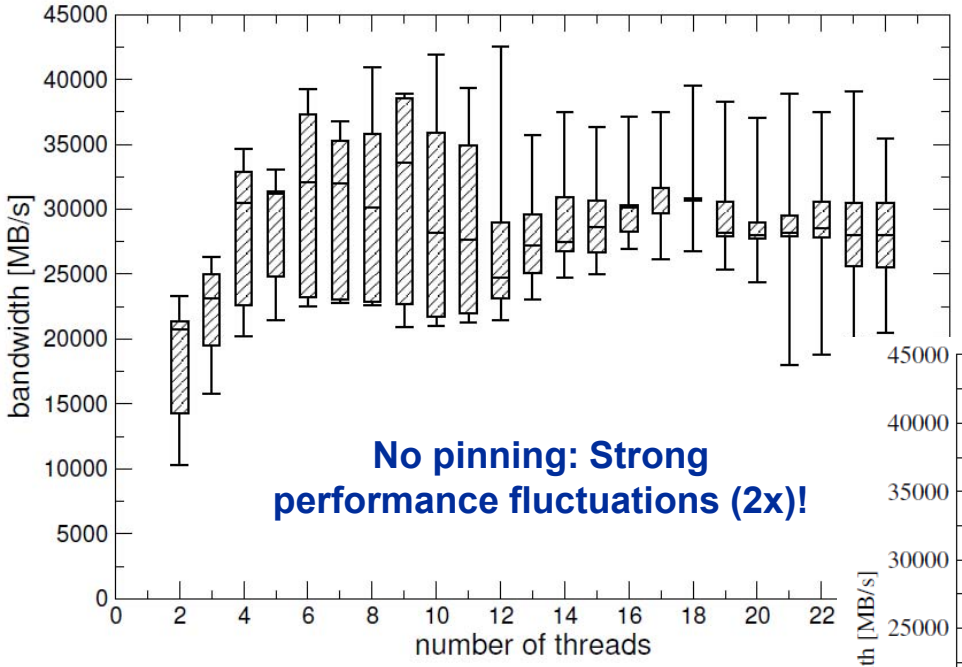
ccNUMA performance problems
Affinity matters

- **ccNUMA:**
  - Whole memory is transparently accessible by all processors
  - but physically distributed
  - with varying bandwidth and latency
  - and potential contention (shared memory paths)
- **How do we make sure that memory access is always as "local" and "distributed" as possible?**

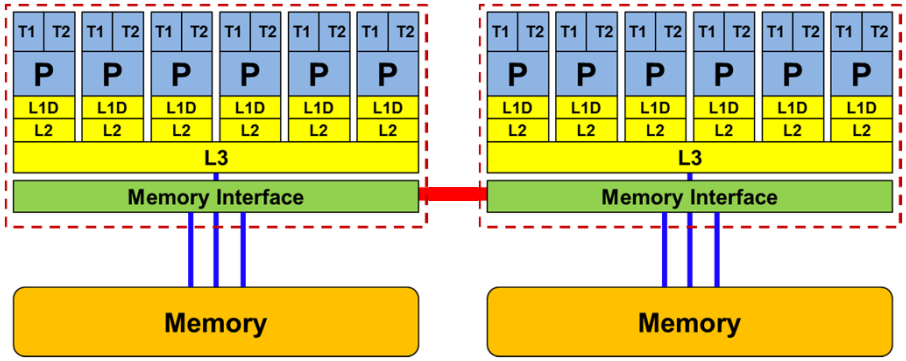


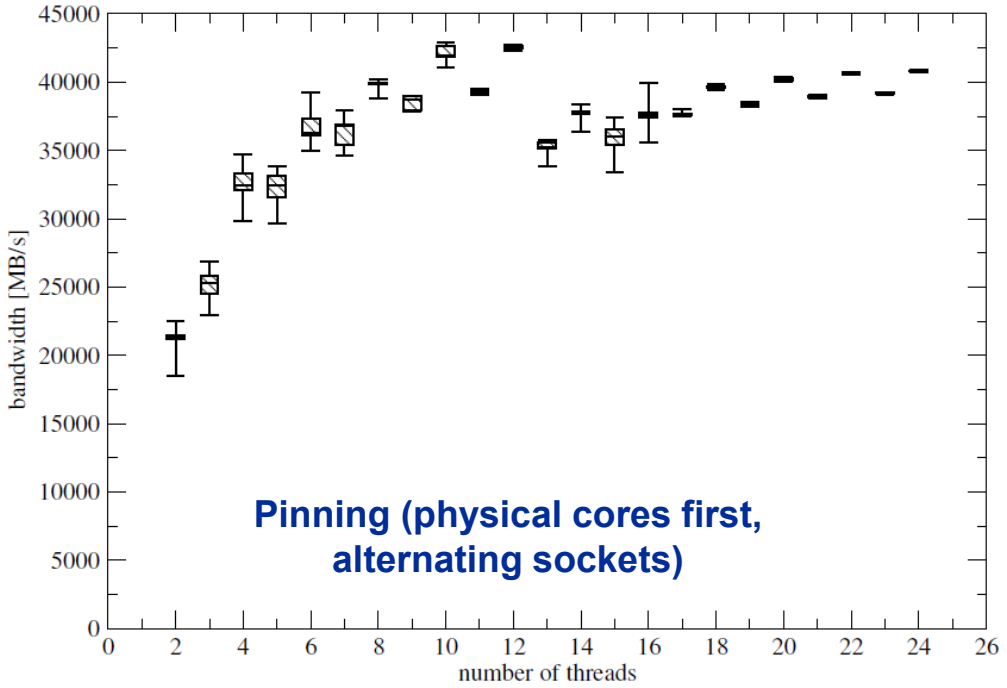
  - Page placement is implemented in units of OS pages (often 4kB, possibly more)
  - Be aware that you are not the only one using memory (even on a dedicated node)

STREAM benchmark on 2x6-core Intel Westmere
Anarchy vs. thread pinning



No pinning: Strong performance fluctuations (2x)!





Pinning (physical cores first, alternating sockets)

Pinning threads on ccNUMA is a must:

- **Eliminating performance variation**
- **Making use of architectural features**
- **Avoiding resource contention**

Erratic performance on ccNUMA nodes

- **Thread pinning is essential!**
- **Data distribution within OpenMP codes must be correctly done by programmer → first touch**
- **Even MPI applications suffer in some cases from data locality issues:**
  - OS allocates (file) buffers → Your MPI performance may depend on the type of the previous jobs!
  - Only system administrator can release the OS buffer at OS level
  - User may use a sweeper code…. (Touch all available memory in the node once then OS releases the buffer)

LIKWID: Lightweight Performance Tools for efficiently using and programming multicores

- **Lightweight command line tools for Linux**
- **Help to face the challenges without getting in the way**
- **Focus on X86 architecture**
- **Philosophy:**
  - Simple
  - Efficient
  - Portable
  - Extensible
- **Get around some some mysteries with LIKWID, e.g. pinning**

Open source project (GPL v2):

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

THANK YOU.

Jan Treibig
Johannes Habich
Moritz Kreutzer
Markus Wittmann
Thomas Zeiser
Michael Meier
Faisal Shahzad
Gerald Schubert

OMI4papps
HQS@HPC II

hpcADD
SKALB

References

Books:

- G. Hager and G. Wellein: Introduction to High Performance Computing for Scientists and Engineers. CRC Computational Science Series, 2010. ISBN 978-1439811924

Papers:

- G. Hager, J. Treibig, J. Habich and G. Wellein: Exploring performance and power properties of modern multicore chips via simple machine models. Accepted. Preprint: arXiv:1208.2908
- J. Treibig, G. Hager and G. Wellein: Performance patterns and hardware metrics on modern multicore processors: Best practices for performance engineering. Workshop on Productivity and Performance (PROPER 2012) at Euro-Par 2012, August 28, 2012, Rhodes Island, Greece. Preprint: arXiv:1206.3738
- M. Kreutzer, G. Hager, G. Wellein, H. Fehske, A. Basermann and A. R. Bishop: Sparse Matrix-vector Multiplication on GPGPU Clusters: A New Storage Format and a Scalable Implementation. Workshop on Large-Scale Parallel Processing 2012 (LSPP12), DOI: 10.1109/IPDPSW.2012.211
- J. Treibig, G. Wellein and G. Hager: Efficient multicore-aware parallelization strategies for iterative stencil computations. Journal of Computational Science 2 (2), 130-137 (2011). DOI 10.1016/j.jocs.2011.01.010

BLOG: http://blogs.fau.de/hager/

References

Papers continued:

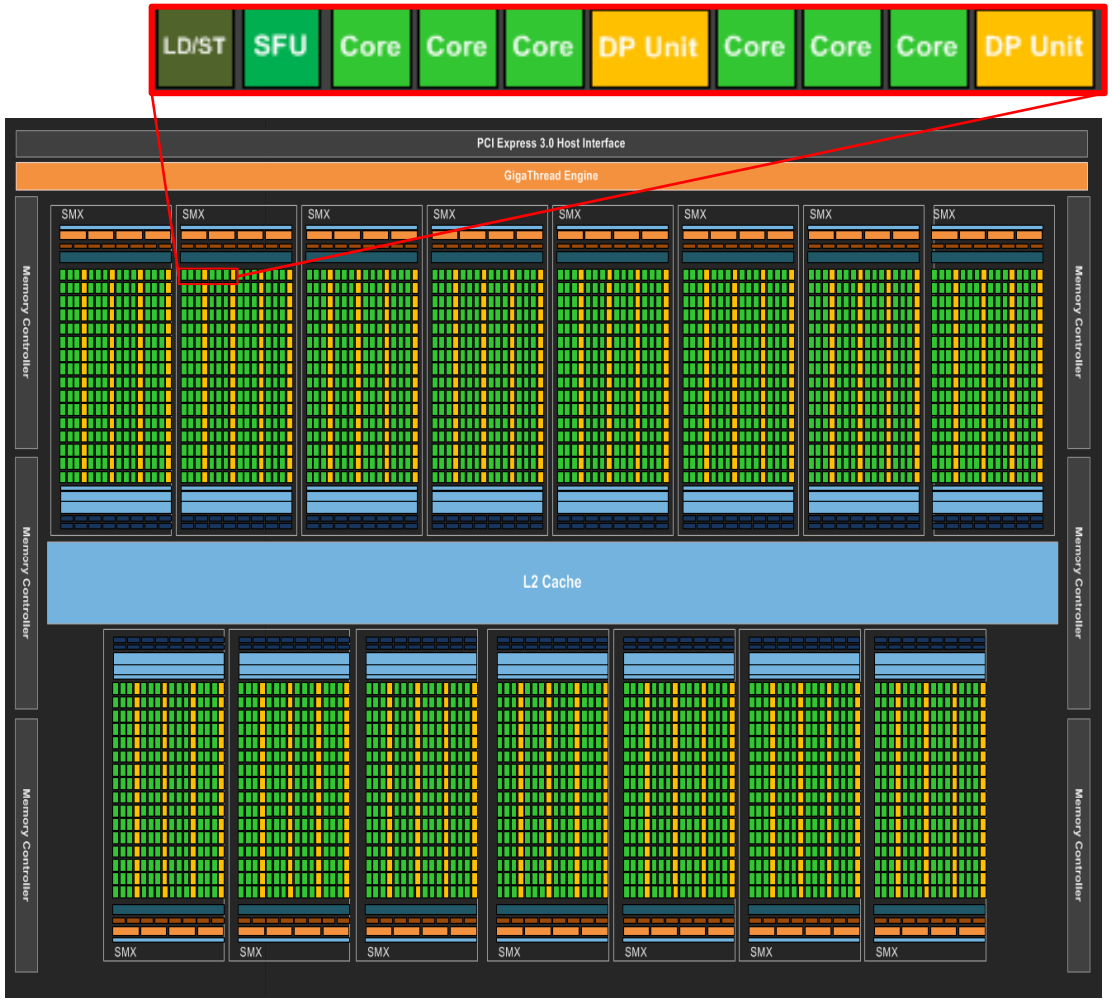
- G. Wellein, G. Hager, T. Zeiser, M. Wittmann and H. Fehske: Efficient temporal blocking for stencil computations by multicore-aware wavefront parallelization. Proc. COMPSAC 2009.
  DOI: 10.1109/COMPSAC.2009.82
- M. Wittmann, G. Hager, J. Treibig and G. Wellein: Leveraging shared caches for parallel temporal blocking of stencil codes on multicore processors and clusters. Parallel Processing Letters **20** (4), 359-376 (2010).
  DOI: 10.1142/S0129626410000296. Preprint: arXiv:1006.3148
- J. Treibig, G. Hager and G. Wellein: LIKWID: A lightweight performance-oriented tool suite for x86 multicore environments. Proc. PSTI2010, the First International Workshop on Parallel Software Tools and Tool Infrastructures, San Diego CA, September 13, 2010.
  DOI: 10.1109/ICPPW.2010.38. Preprint: arXiv:1004.4431
- G. Schubert, H. Fehske, G. Hager, and G. Wellein: Hybrid-parallel sparse matrix-vector multiplication with explicit communication overlap on current multicore-based systems. Parallel Processing Letters 21(3), 339-358 (2011).
  DOI: 10.1142/S0129626411000254
- J. Treibig, G. Wellein and G. Hager: Efficient multicore-aware parallelization strategies for iterative stencil computations. Journal of Computational Science 2 (2), 130-137 (2011).
  DOI 10.1016/j.jocs.2011.01.010

NVIDIA Kepler GK110 Block Diagram

Architecture

- 7.1B Transistors
- 15 “SMX” units
  - 192 (SP) “cores” each
- > 1 TFLOP DP peak
- 1.5 MB L2 Cache
- 384-bit GDDR5
- PCI Express Gen3

- 3:1 SP:DP performance



Intel Xeon Phi block diagram

Architecture

- 3B Transistors
- 60+ cores
- 512 bit SIMD
- ≈ 1 TFLOP DP peak
- 0.5 MB L2/core
- GDDR5
- 2:1 SP:DP performance