MAX PLANCK
COMPUTING & DATA FACILITY

MPCDF

HIGH-PERFORMANCE COMPUTING TECHNOLOGY HARDWARE AND SOFTWARE TRENDS AND RESULTING CHALLENGES

Markus Rampp (MPCDF)

NOMAD Summer, Paphos/Cyprus, Oct 3, 2023

CPU: X86_64

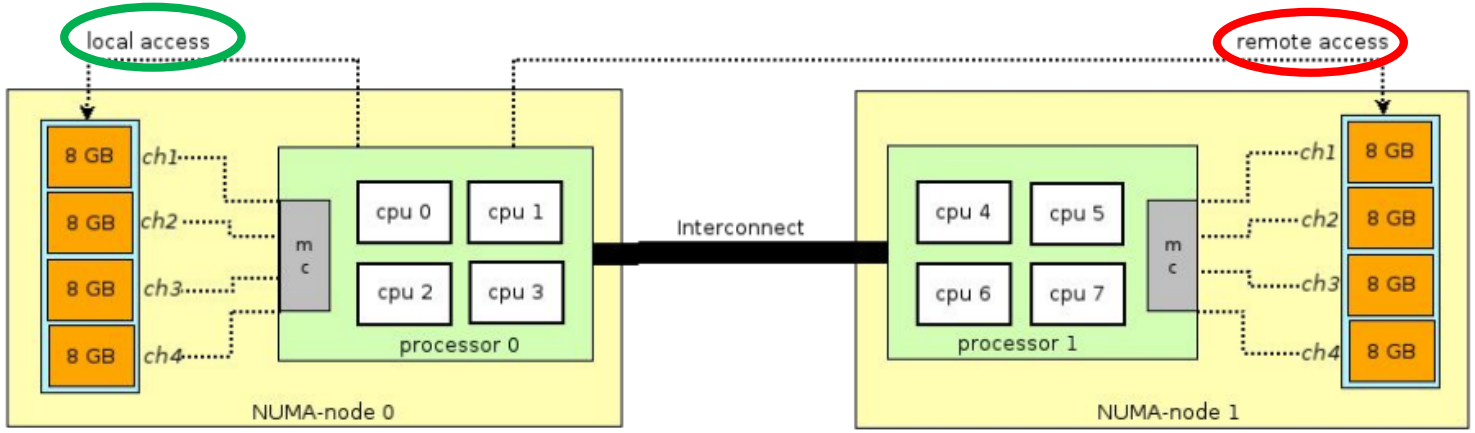
CPU-architecture:

- **2 CPUs („sockets“) per HPC node (*note: all numbers below are per CPU*), increasing core counts**
- **traditionally: 1 „socket“ == 1 NUMA domain (homogeneous access to all cores)**

-> Intel Xeon adopts „chiplet“ concept of AMD EPYC for manufacturing reasons

=> new packaging opportunities

=> NUMA sub-structure (8…16 cores per chiplet)



CPU: NUMA SUBSTRUCTURE IN X86_64

AMD with EPYC CPUs introduced „chiplets“ for manufacturing reasons, adopted by Intel Xeon (2023)

```
Intel-Xeon-Icelake:~$ numactl --hardware
available: 2 nodes (0-1)
node 0 cpus: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
 32 33 34 35 72 73 74 75 76 77 78 79 80 81 82 83
 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99
 100 101 102 103 104 105 106 107
[...]
node distances:
node   0   1
  0:  10  20
  1:  20  10
```

```
AMD-EPYC-Rome:~$ numactl --hardware
available: 8 nodes (0-7)
node 0 cpus: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
[...]
node distances:
node   0   1   2   3   4   5   6   7
  0:  10  12  12  12  32  32  32  32
  1:  12  10  12  12  32  32  32  32
  2:  12  12  10  12  32  32  32  32
  3:  12  12  12  10  32  32  32  32
  4:  32  32  32  32  10  12  12  12
  5:  32  32  32  32  12  10  12  12
  6:  32  32  32  32  12  12  10  12
  7:  32  32  32  32  12  12  12  10
```

CPU: NUMA SUBSTRUCTURE IN X86_64

Beware of inconsistent notation: *node* means "NUMA-node" (socket, chiplet, …) not cluster node here !

AMD with EPYC CP oduced „chiplets" for manufacturing reasons, adopted by Intel Xeon (2023)

```
Intel-Xeon-Icelake:~$ numactl --hardware
available: 2 nodes (0-1)
node 0 cpus: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
 32 33 34 35 72 73 74 75 76 77 78 79 80 81 82 83
 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99
 100 101 102 103 104 105 106 107
[...]
node distances:
node   0   1
  0:  10  20
  1:  20  10
```

```
AMD-EPYC-Rome:~$ numactl --hardware
available: 8 nodes (0-7)
node 0 cpus: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
[...]
node distances:
node   0   1   2   3   4   5   6   7
  0:  10  12  12  12  32  32  32  32
  1:  12  10  12  12  32  32  32  32
  2:  12  12  10  12  32  32  32  32
  3:  12  12  12  10  32  32  32  32
  4:  32  32  32  32  10  12  12  12
  5:  32  32  32  32  12  10  12  12
  6:  32  32  32  32  12  12  10  12
  7:  32  32  32  32  12  12  12  10
```

CPU: X86_64

Architectural trends:

- **increasing core counts approaching ~ 90 (Intel Xeon) to ~128 (AMD EPYC) per CPU**

  … remember „many-core ?“

- **stagnating SIMD width („vector length“) at 512 bit (8-wide FP64 lanes, + FMA, …)**
- **memory capacity & bandwidth (now DDR5) compensating for higher core counts**
- **option for (additional) high-bandwidth memory (~ 128 GB, 1 TB/s)**

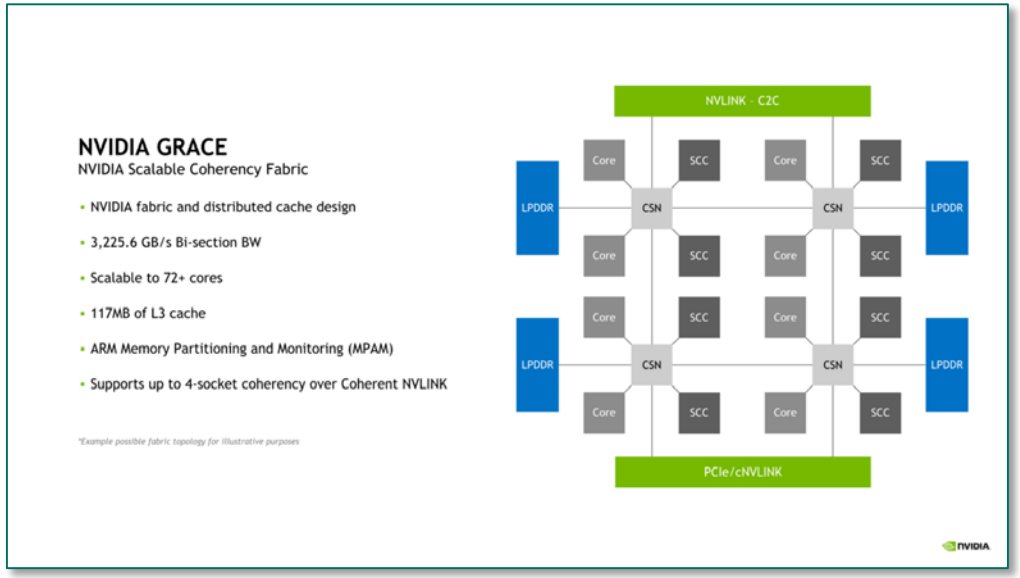
Intel Xeon: Sapphire Rapids (2023): 8 DDR5 memory channels (+HBM option), 50…60 cores

AMD EPYC: Genoa (2022): 12 DDR5 memory channels, 64…96 cores

CPU: ARM

Nvidia GRACE:

- 72 Armv9 CPU cores with SVE2, LPDDR5



https://www.nvidia.com/en-us/data-center/grace-cpu-superchip/

SiPEARL (European Processor Initiative)

- focus on memory bandwidth (HBM) but timeline, funding, … ?

Other:

- Cavium/Marvell ThunderX (Isambard) looks dead ?
- A64FX (Fugaku): did not make it into the market outside Japan
- Apple M1 et al. (not yet ready for FP64 but looks interesting, cf. https://arxiv.org/abs/2211.00720)

GPU: EVOLUTION AND DIVERSIFICATION OF VENDORS

Typical node architecture:

- **4 GPUs per HPC node, attached via PCIe5 (64 GB/s per GPU) to „host“ CPUs**
- **fast intra-node GPU network, e.g. Nvlink4 (450 GB/s per GPU)**
- **optional: “direct“ communication paths**

  from GPUs to network (e.g. LUMI)

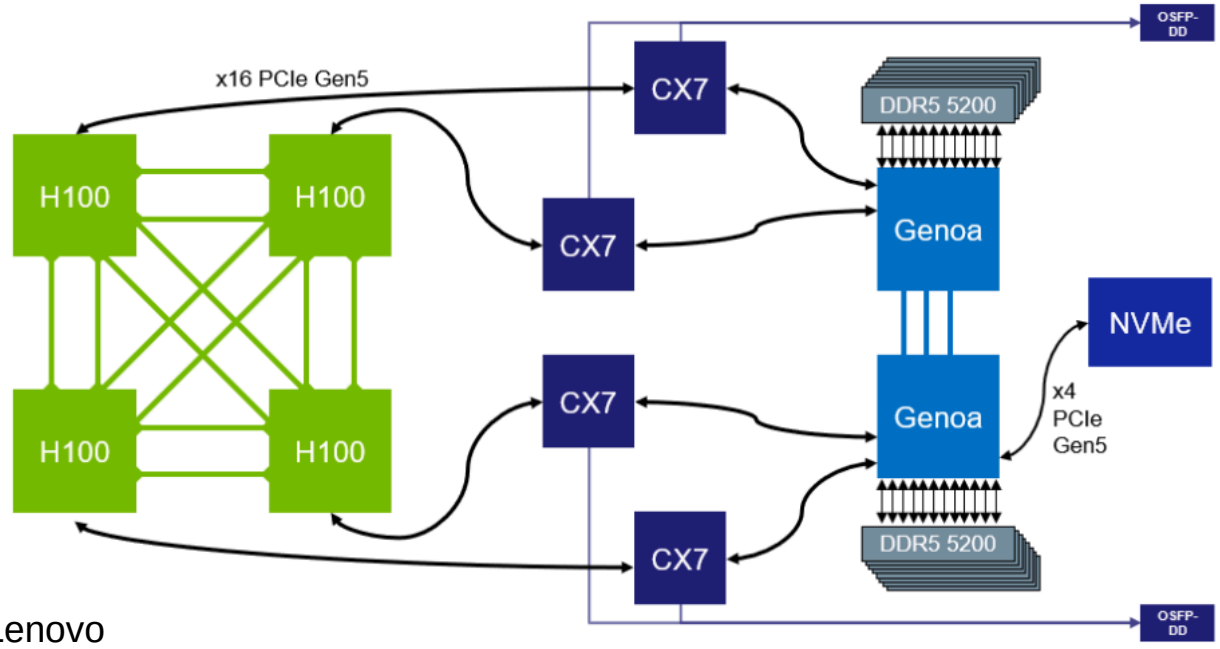


Image credit: M. Hiegl, Lenovo

GPU: EVOLUTION AND DIVERSIFICATION OF VENDORS

GPU models:

- **Nvidia „Hopper“ H100 (2023): evolution of A100: 96 GB HBM, 34 TF/s (FP64), 3.3 TB/s @600W**
- **AMD Instinct MI250 (2022): evolution of MI100: 128 GB HBM, 45 TF/s (FP64), 3.3 TB/s @500W**
- **Intel Ponte Vecchio (202?):**

GPU: EVOLUTION AND DIVERSIFICATION OF VENDORS

GPU models:

- **Nvidia „Hopper“ H100 (2023): evolution of A100: 96 GB HBM, 34 TF/s (FP64), 3.3 TB/s @600W**
- **AMD Instinct MI250 (2022): evolution of MI100: 128 GB HBM, 45 TF/s (FP64), 3.3 TB/s @500W**
- **Intel Ponte Vecchio (202?):**

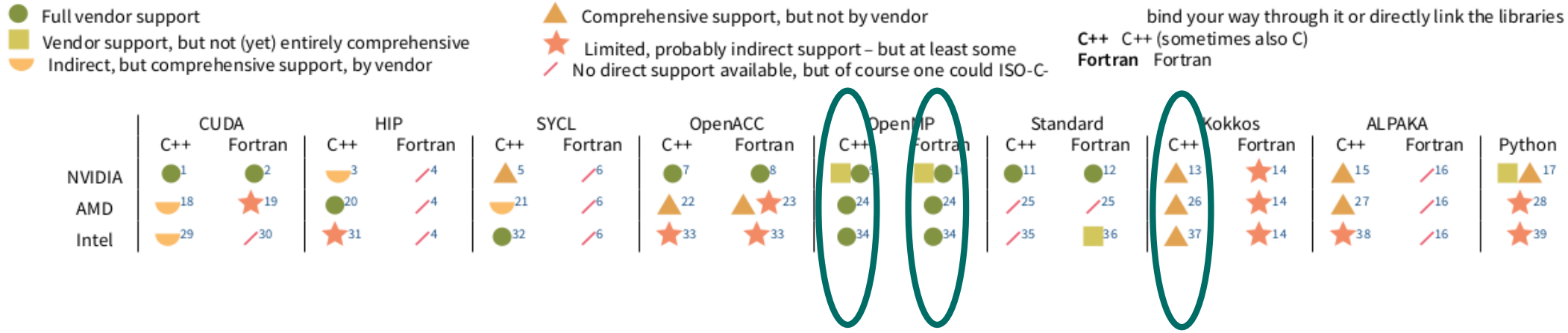
For comparisons beware (a consequence of modularization cf. „chiplets“):

what is a single „GPU“?

- **AMD MI250 GPU is effectively a complex of 2 MI210 GPUs (no transparent shared-memory !)**
- **Intel PonteVecchio GPU is (expected to be) a complex of 4-8(?) „tiles“ (shared-memory ?)**

=> *benchmarks/comparisons on a per Watt (TDP) basis*

GPU PROGRAMMING MODELS - COMPATIBILITY



by Andreas Herten (JSC), https://arxiv.org/abs/2309.05445

GPU PROGRAMMING MODELS - COMPATIBILITY

- 1: CUDA C/C++ is supported on NVIDIA GPUs through the CUDA Toolkit
- 2: CUDA Fortran, a proprietary Fortran extension, is supported on NVIDIA GPUs via the NVIDIA HPC SDK
- 3: HIP programs can directly use NVIDIA GPUs via a CUDA backend; HIP is maintained by AMD
- 4: No such thing like HIP for Fortran
- 5: SYCL can be used on NVIDIA GPUs with *experimental* support either in SYCL directly or in DPC++, or via hipSYCL
- 6: No such thing like SYCL for Fortran
- 7: OpenACC C/C++ supported on NVIDIA GPUs directly (and best) through NVIDIA HPC SDK; additional, somewhat limited support by GCC C compiler and in LLVM through Clacc
- 8: OpenACC Fortran supported on NVIDIA GPUs directly (and best) through NVIDIA HPC SDK; additional, somewhat limited support by GCC Fortran compiler and Flacc
- 9: OpenMP in C++ supported on NVIDIA GPUs through NVIDIA HPC SDK (albeit with a few limits), by GCC, and Clang; see OpenMP ECP BoF on status in 2022.
- 10: OpenMP in Fortran supported on NVIDIA GPUs through NVIDIA HPC SDK (but not full OpenMP feature set available), by GCC, and Flang
- 11: pSTL features supported on NVIDIA GPUs through NVIDIA HPC SDK
- 12: Standard Language parallel features supported on NVIDIA GPUs through NVIDIA HPC SDK
- 13: Kokkos supports NVIDIA GPUs by calling CUDA as part of the compilation process
- 14: Kokkos is a C++ model, but an official compatibility layer (*Fortran Language Compatibility Layer*, FLCL) is available.
- 15: Alpaka supports NVIDIA GPUs by calling CUDA as part of the compilation process
- 16: Alpaka is a C++ model
- 17: There is a vast community of offloading Python code to NVIDIA GPUs, like CuPy, Numba, cuNumeric, and many others; NVIDIA actively supports a lot of them, but has no direct product like *CUDA for Python*; so, the status is somewhere in between
- 18: hipify by AMD can translate CUDA calls to HIP calls which runs natively on AMD GPUs
- 19: AMD offers a Source-to-Source translator to convert some CUDA Fortran functionality to OpenMP for AMD GPUs (gpufort); in addition, there are ROCm library bindings for Fortran in hipfort
  OpenACC/CUDA Fortran Source-to-Source translator
- 20: HIP is the preferred native programming model for AMD GPUs
- 21: SYCL can use AMD GPUs, for example with hipSYCL or DPC++ for HIP AMD
- 22: OpenACC C/C++ can be used on AMD GPUs via GCC or Clacc; also, Intel's OpenACC to OpenMP Source-to-Source translator can be used to generate OpenMP directives from OpenACC directives
- 23: OpenACC Fortran can be used on AMD GPUs via GCC; also, AMD's gpufort Source-to-Source translator can move OpenACC Fortran code to OpenMP Fortran code, and also Intel's translator can work
- 24: AMD offers a dedicated, Clang-based compiler for using OpenMP on AMD GPUs: AOMP; it supports both C/C++ (Clang) and Fortran (Flang, example)
- 25: Currently, no (known) way to launch Standard-based parallel algorithms on AMD GPUs
- 26: Kokkos supports AMD GPUs through HIP
- 27: Alpaka supports AMD GPUs through HIP
- 28: AMD does not officially support GPU programming with Python (also not semi-officially like NVIDIA), but third-party support is available, for example through Numba (currently inactive) or a HIP version of CuPy
- 29: SYCLomatic translates CUDA code to SYCL code, allowing it to run on Intel GPUs; also, Intel's DPC++ Compatibility Tool can transform CUDA to SYCL
- 30: No direct support, only via ISO C bindings, but at least an example can be found on GitHub; it's pretty scarce and not by Intel itself, though
- 31: CHIP-SPV supports mapping CUDA and HIP to OpenCL and Intel's Level Zero, making it run on Intel GPUs
- 32: SYCL is the prime programming model for Intel GPUs; actually, SYCL is only a standard, while Intel's implementation of it is called DPC++ (*Data Parallel C++*), which extends the SYCL standard in various places; actually actually, Intel namespaces everything *oneAPI* these days, so the *full* proper name is Intel oneAPI DPC++ (which incorporates a C++ compiler and also a library)
- 33: OpenACC can be used on Intel GPUs by translating the code to OpenMP with Intel's Source-to-Source translator
- 34: Intel has extensive support for OpenMP through their latest compilers
- 35: Intel supports pSTL algorithms through their DPC++ Library (oneDPL; GitHub). It's heavily namespaced and not yet on the same level as NVIDIA
- 36: With Intel oneAPI 2022.3, Intel supports DO CONCURRENT with GPU offloading
- 37: Kokkos supports Intel GPUs through SYCL
- 38: Alpaka v0.9.0 introduces experimental SYCL support
- 39: Not a lot of support available at the moment, but notably DPNP, a SYCL-based drop-in replacement for Numpy, and numba-dpex, an extension of Numba for DPC++.

Note the fine print !

by Andreas Herten (JSC), https://arxiv.org/abs/2309.05445

GPU: ARCHITECTURAL TRENDS

Tighter integration of CPU and GPU

- **Nvidia „Grace-Hopper“: cache-coherent memory (DDR, HBM) based on fast chip-to-chip interconnect & ATS**

coming 2023 in ALPS (CSCS)

- **AMD APU „MI300…“ (single HBM) based on CPU and GPU „chiplets“ and on-chip access to a single high-bandwidth memory space**

coming 2023 in El Capitan (LLNL)

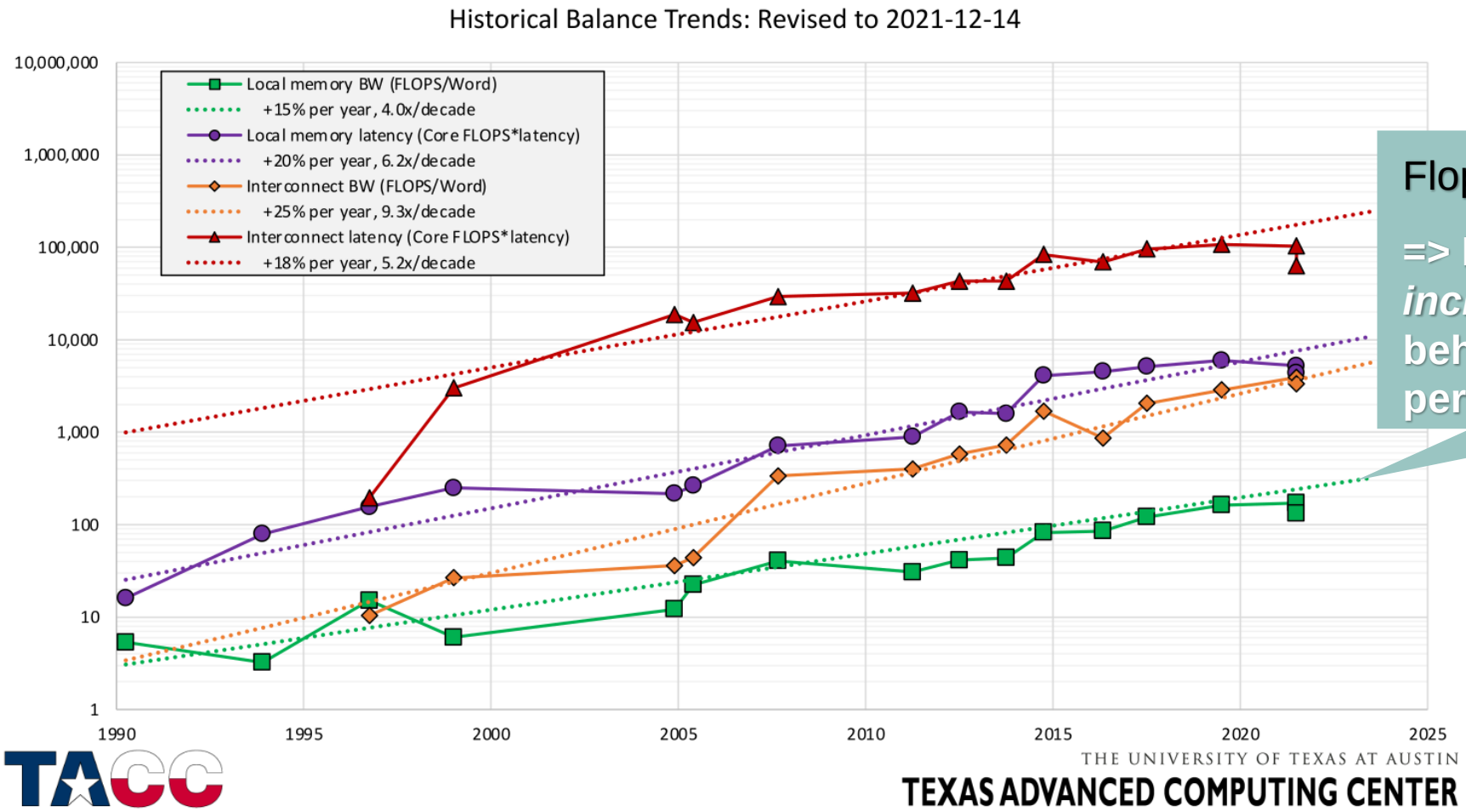
coming 2024 in Viper (MPCDF)

=> programming opportunities for HPC codes

will look essentially like fast „unified (virtual) memory“

CPU and GPU exposed by OS as NUMA nodes

TRENDS: SYSTEM BALANCE



Flop/s / Byte/s = Flop/Byte

=> Memory bandwidth is *increasingly* lagging behind floating point performance

THE UNIVERSITY OF TEXAS AT AUSTIN
TEXAS ADVANCED COMPUTING CENTER

John McCalpin: Memory Bandwidth and System Balance in HPC Systems (SC'16), update taken from:https://hpc.fau.de/files/2021/12/memorybw_systembalance_slides_2021-12-15.pdf

SYSTEM BALANCE: THE “MEMORY WALL“

Roofline model:

- memory performance evolution [GB/s] lags behind floating-point performance [GFlop/s]:

  => achievable performance is increasingly bound by memory bandwidth (not FP-performance)

- use **algorithmic intensity, AI [Flops/Byte]**, to classify algorithm (kernels) **and relate to hardware limits**

  => can prevent from futile overoptimization attempts of memory-bandwidth bound algorithms

Performance R < Flops/Byte * Bytes/s

Algorithmic Intensity, ***AI (algorithm)***

memory performance, ***R_{Mem} , (hardware)***

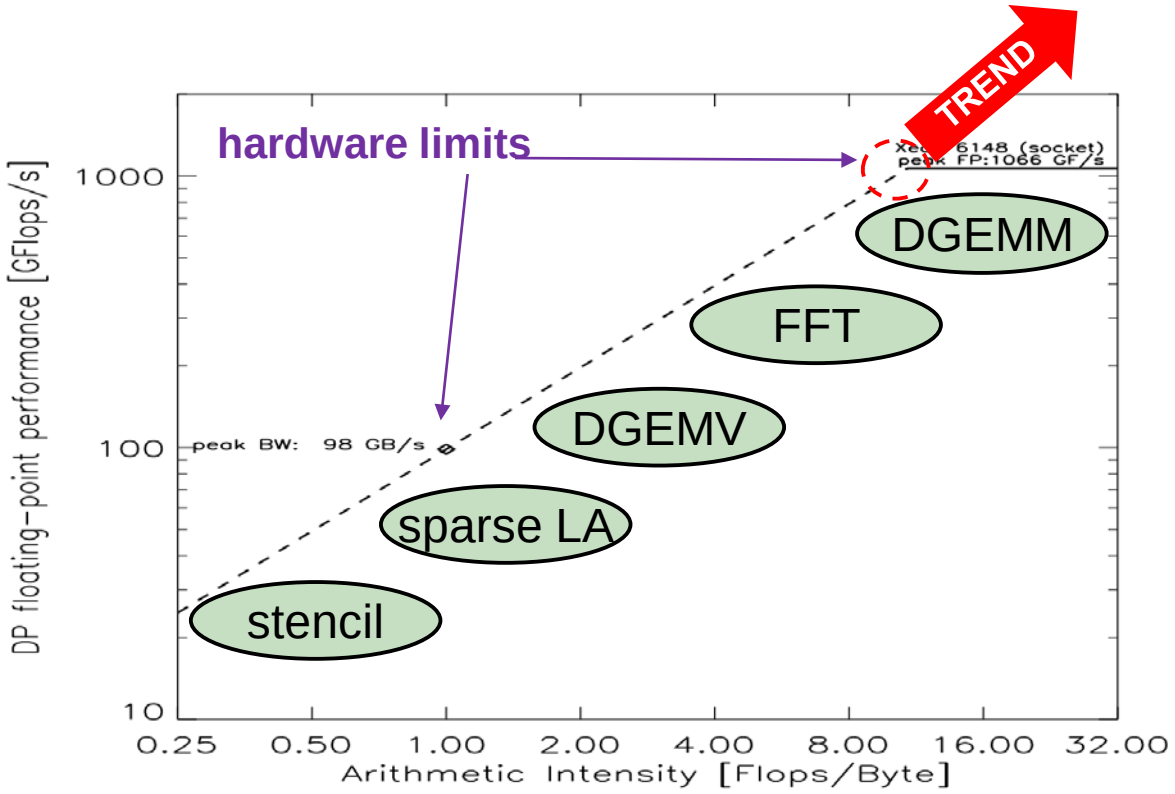
$$R < \min(R_{FP}, AI * R_{Mem})$$

(S. Williams et al. Roofline: an insightful visual performance model for multicore architectures, Comm. ACM, 2009)

PROCESSOR BALANCE: CPU AND GPU

Recall roofline model: performance R < Flops/Byte * Byte/s

algorithm *memory performance*



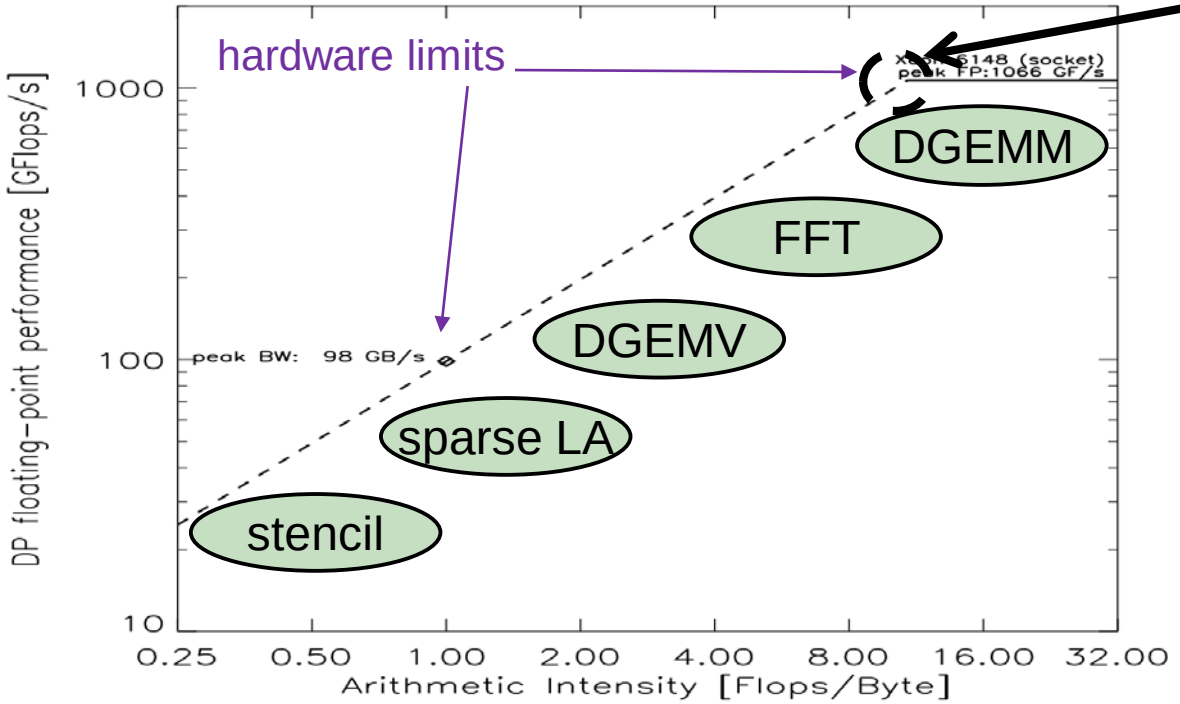
examples ("double precision" = 8 Bytes/element):

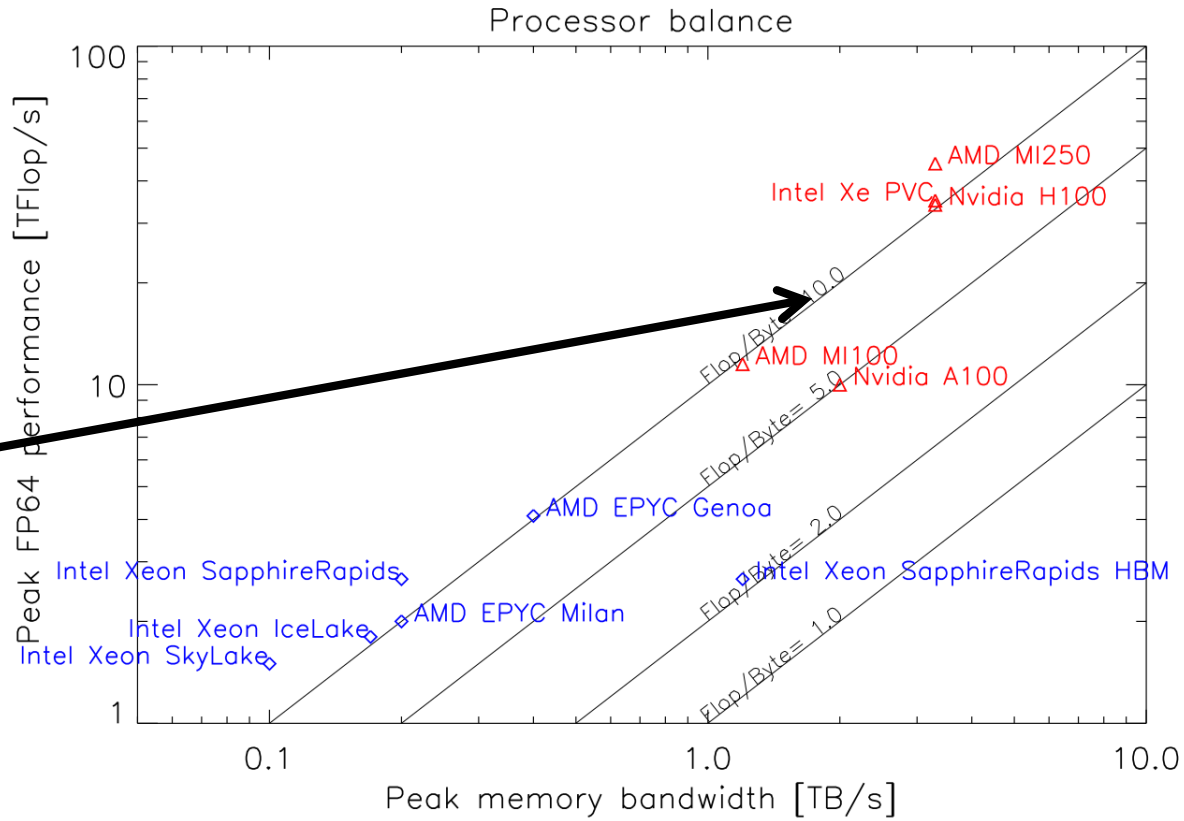
- **x(i)=a(i)+c*b(i)**
  => AI=(2 ops)/(3*8 Bytes)=0.08
- **r(i)=x(i)**2+y(i)**2**
  => AI=(3 ops)/(3*8 Bytes)~0.125
- **DGEMV** (BLAS2): $O(N^2)$ ops on $O(N^2)$ Bytes
  => AI ~ 2
- **DGEMM** (BLAS3): $O(N^3)$ ops on $O(N^2)$ Bytes
  => AI =O(N)
- **FFT**: O(N log N) ops on O(N) Bytes
  => AI = O(log(N))

PROCESSOR BALANCE: CPU AND GPU

Recall roofline model: performance R < Flops/Byte * Bytes/s

algorithm *memory performance*





note: some numbers are based on speculations

COUNTERMEASURES

Algorithmic (increase algorithmic intensity)

- more compute on less data (learn the new “Flops are cheap” paradigm)

  ⇨ high-order algorithms (e.g. high-order Discontinuous Galerkin methods in CFD)

Technological: high-bandwidth memory (HBM)

- GPU (>1TB/s), CPU (Xeon with HBM, EPYC with DRAM), ARM ?

  ⇨ comes at a price: HBM is expensive and scarce (~ 100 GB)

PROGRAMMING (PRE-)EXASCALE ARCHITECTURES (1)

MPI+X remains as the prevalent programming model for CPUs

Mostly „business as usual“ in the CPU world (x86_64 and likely also ARM with SVE):

- **increasing core counts motivates intra-node shared-memory programming X=OpenMP, MPI-3, …**
- **sub-NUMA chiplet structure:**
  - **OpenMP domain size stagnates, no excessive shared-memory/thread-scaling**

⇒ **for hybrid MPI/OpenMP codes run 1 MPI rank per NUMA domain (can be 8…16 ranks per node)**

HYBRID PROGRAMMING MODEL

https://computing.llnl.gov/tutorials/parallel_comp/#MemoryArch

in practice**: MPI + X** with X ϵ {**OpenMP**, pthreads, **MPI-3 shared memory**, PGAS, ...}

most common and natural for X: thread-based programming and parallelization model

- OpenMP (high level directives and functions, standardized, C, C++, FORTRAN)
- OpenACC (high level directives and functions, standardized, C, C++, FORTRAN)
- alternatives: CUDA, HIP, SYCL, pthreads (low level, explicit, C only), Intel TBB (evolving?), …

→ OpenMP 4.5 and later as a *possible* convergence of CPU and GPU programming models

HYBRID PROGRAMMING MODEL

Motivation and expected advantages of mixing MPI with a shared-memory model

fewer MPI-ranks (**up to factor 1/cores_per_node ; in practice: 1/cores_per_numadomain**

- reduces number of connections by 2-3 orders of magnitude, e.g. all-to-all: $N^2 \rightarrow (N/20)^2$
- contain MPI initialization times (can be significant for $O(10^4)$ MPI tasks and MPI_alltoall)
- reduce latency overhead: fewer, larger messages, e.g. point-to-point
- reduce memory footprint: MPI buffers
- reduce network contention

HYBRID PROGRAMMING MODEL

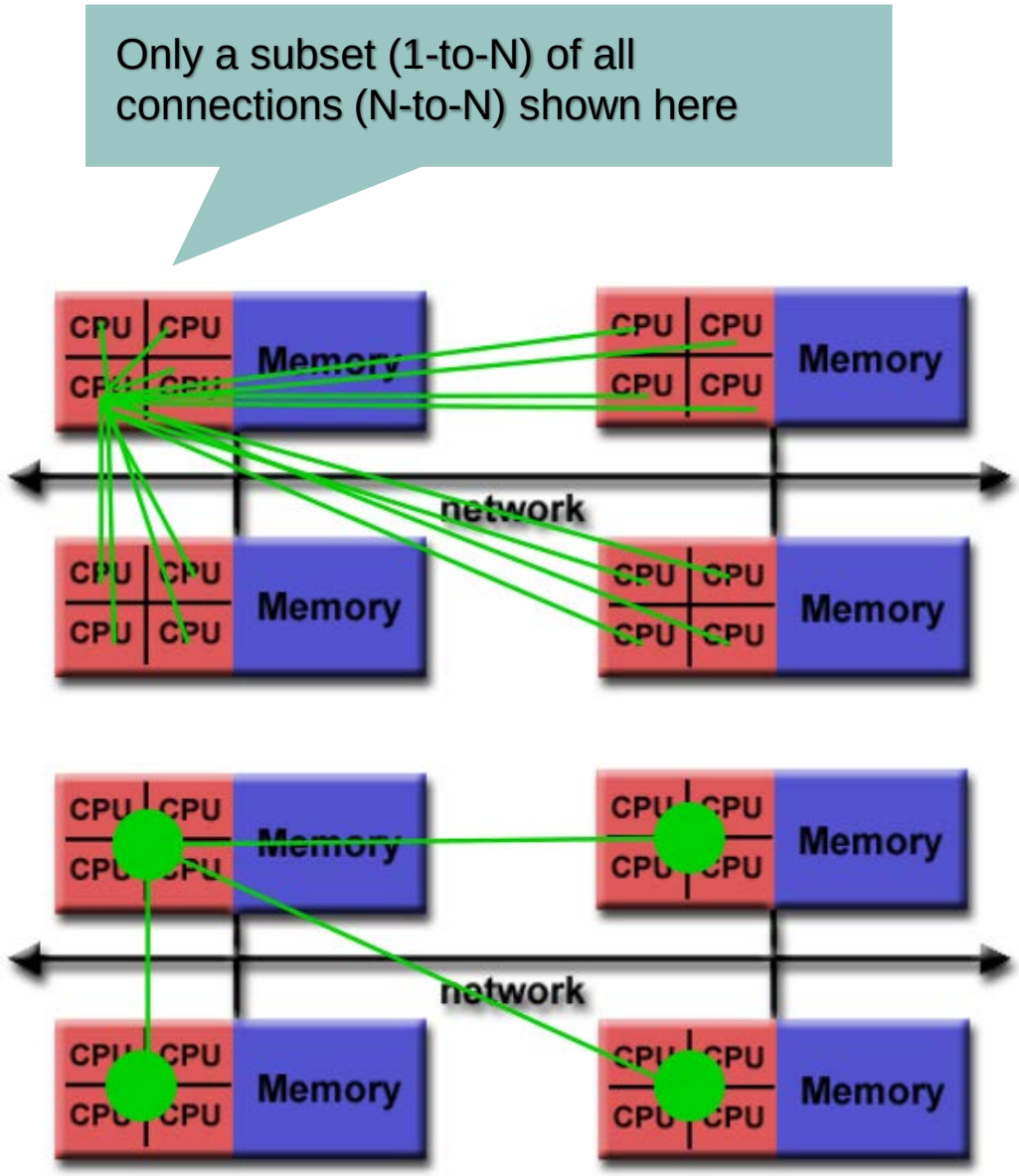
Collective Communication (ex. All-to-All)

Number of connections for **N CPU cores**:
N*(N-1)/2 = 16*15/2 = 120

(NB: MPI library can optimize node-internal traffic)

Number of connections for **N/4 CPU nodes**:
N/4*(N/4-1)/2 = 16*15/2 = 6

(NB: collective reductions can be offloaded to network hardware)



HYBRID PROGRAMMING MODEL

Motivation and expected advantages of mixing MPI with a shared-memory model

application-specific:

- reduce memory footprint by sharing of common data (e.g. read-only tables)
- reduce MPI-communication/computation ratio, e.g. smaller surface-to-volume ratio
- increase size of data domains and use “dynamic” thread scheduling to mitigate load-imbalances, OpenMP has built-in load-balancing mechanisms e.g.:

```
$!OMP PARALLEL DO SCHEDULE(DYNAMIC)
```

- use highly efficient threaded BLAS (e.g. MKL) or higher linear algebra

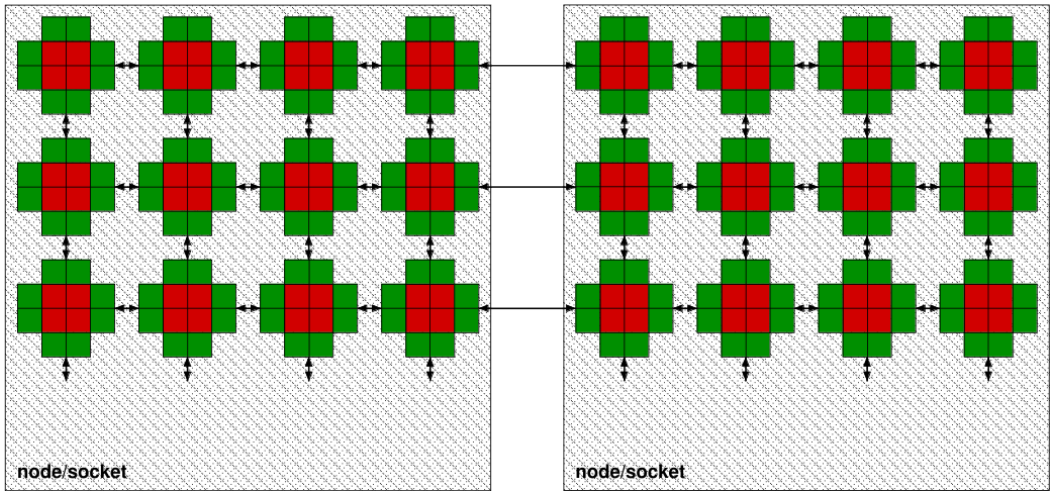
HYBRID PROGRAMMING EXAMPLE

Example: Domain Decomposition

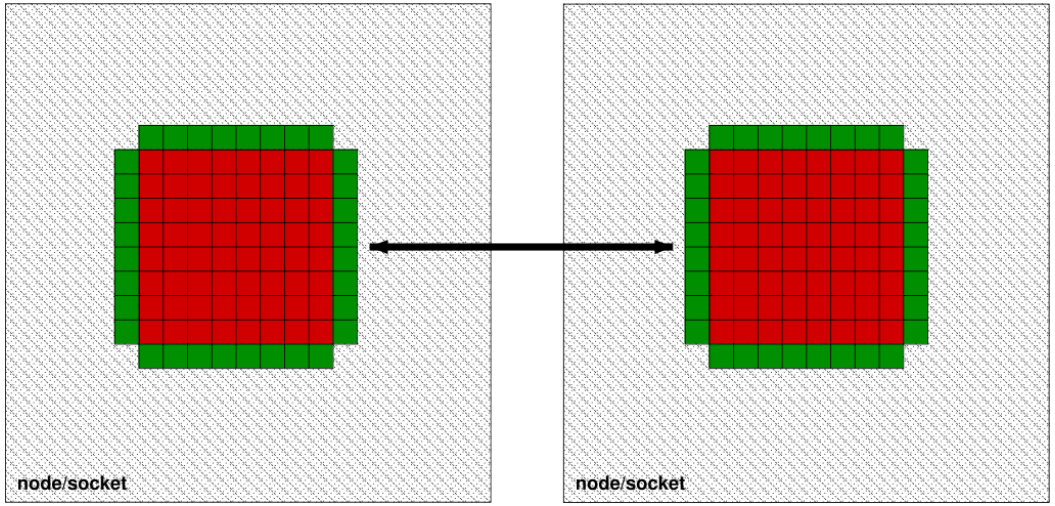
- **halo-exchange (for stencil computation)**
- **explicit message passing (MPI next-neighbour)**
- **general concept:**

  coarse-grained data decomposition to mitigate latencies/synchronization: long computation phases with short communication

Plain MPI



Hybrid: e.g. MPI + OpenMP or MPI + MPI (shared memory)



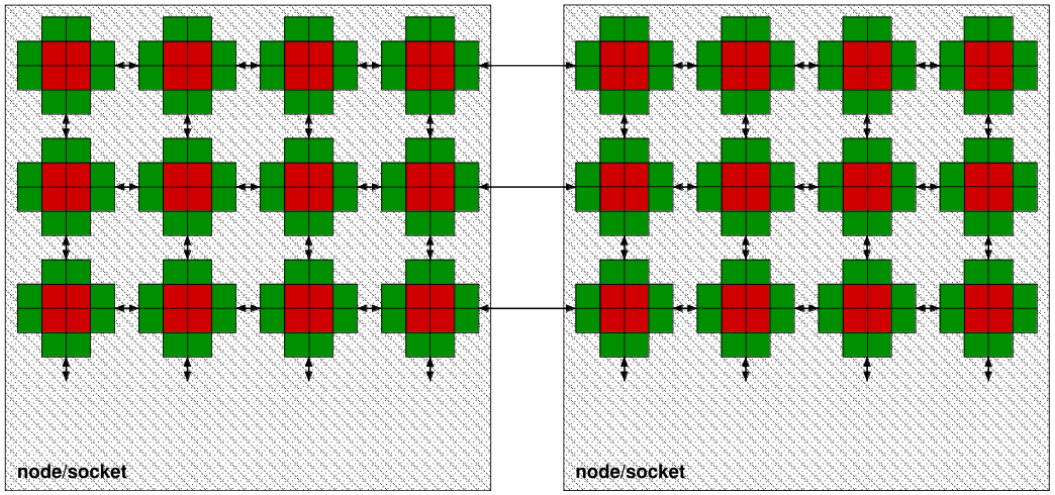
HYBRID PROGRAMMING EXAMPLE

Example: Domain Decomposition

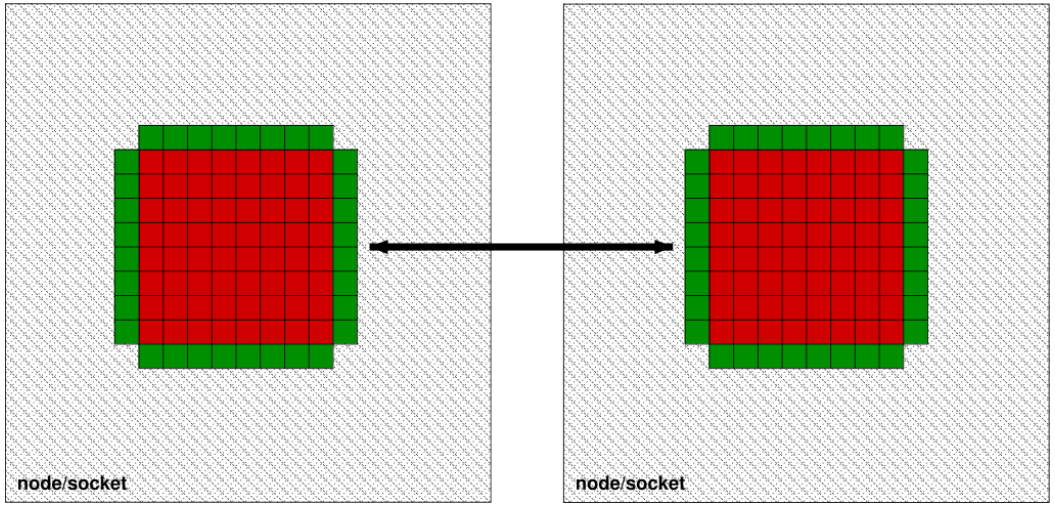
Advantages:

- **fewer MPI-ranks: * 1/cores per node (can be very beneficial for all-to-all type messaging)**
- **fewer but larger messages → reduce latency**
- **smaller surface/volume ~ communication/computation**
- **smaller buffer size**
- **sharing of common data (e.g. tables)**
- **mitigate load-imbalances (e.g. using OpenMP non-static loop schedules) within (large) domain**

Plain MPI



Hybrid: e.g. MPI + OpenMP or MPI + MPI (shared memory)



PROGRAMMING (PRE-)EXASCALE ARCHITECTURES (2)

MPI+X remains as the prevalent programming model for GPUs

1 (or more) MPI ranks per GPU, exploit Nvlink (via MPI), MPS, …

Tighter CPU-GPU integration (Nvidia Grace-Hopper, AMD MI300A)

- **will likely lower the bar for (incrementally) porting CPU codes:**
  - like the current „unified memory“ concept of Nvidia GPU (transparent data transfers via page faults), but *really fast (latency!)*
- **maybe not a game-changer for GPU applications already optimized for data-locality**

OUTLOOK

- long-term predictions (beyond 2-3 years) about evolution of (HPC) technology are notoriously difficult
- commercial AI applications and requirements of hyperscalers (AWS, …) are major drivers these days
- ***massively parallel, heterogeneous architectures won‘t go away anytime soon***
- ***how to navigate the programming jungle?***

OpenMP SYCL RAJA OpenACC Directives for Accelerators

NVIDIA CUDA julia AMD ROCm alpaka

Be pragmatic: we are always taking a risk – or do nothing and wait for something to save us the day

MY OWN VIEW: *WHAT IF I HAD TO …*

… develop a new (GPU) code ?

use C++ (+ Kokkos, …)

… port an existing CPU Fortran code to GPU ?

use OpenMP

… port an existing CUDA code to multiple GPU platforms ?

hippify for AMD GPUs (and see what happens with other GPU vendors)

… develop or port a CPU code or library with highest ambition for performance and longevity ?

use vendor-specific language (CUDA, HIP, SYCL) encapsulated by software-abstration layer

watch out for consolidation opportunities (the SYCL promise, OpenMP, …)

SOME TECHNICAL REFERENCES

- *Introduction to High Performance Computing for Scientists and Engineers*, G. Hager & G. Wellein, CRC Press, 2011
- C. Märtin, Post-Dennard scaling and the final years of Moore's law, tech. report, 2014
- *What Every Programmer Should Know About Memory*, U. Drepper, 2007
- *Is Parallel Programming Hard, And, If So, What Can You Do About It?*, Paul E. McKenney, 2021 (Second Edition), https://arxiv.org/pdf/1701.00854.pdf
- *Argonne Trainings on Extreme-Scale Computing*: https://extremecomputingtraining.anl.gov/
- NV CUDA Documentation: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html
- Agner Fog's CPU writeups, e.g. https://www.agner.org/optimize/microarchitecture.pdf
- *Technische Informatik 1, 2, and 3*; W. Schiffmann & R. Schmitz, Springer