Parallel Computing Techniques

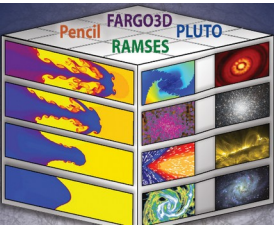
for
Astrophysicists

Seyed Mohammad Hosseinirad

Sheikh Bahai National HPC Center (NHPCC)

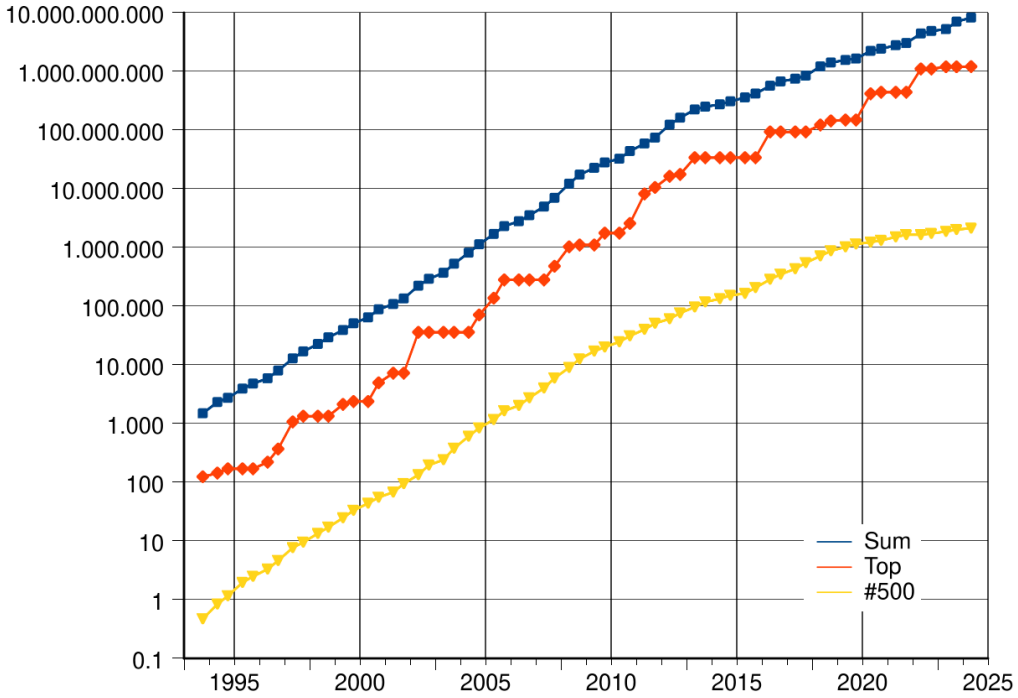
1404 Ordibehesht 11th

A Computational Look at Astrophysical Flows Workshop, School of Astronomy, IPM



Rapid growth of supercomputers

Blue: Combined performance of 500 largest
Red: Fastest ones
Yellow: Supercomputer in 500th place

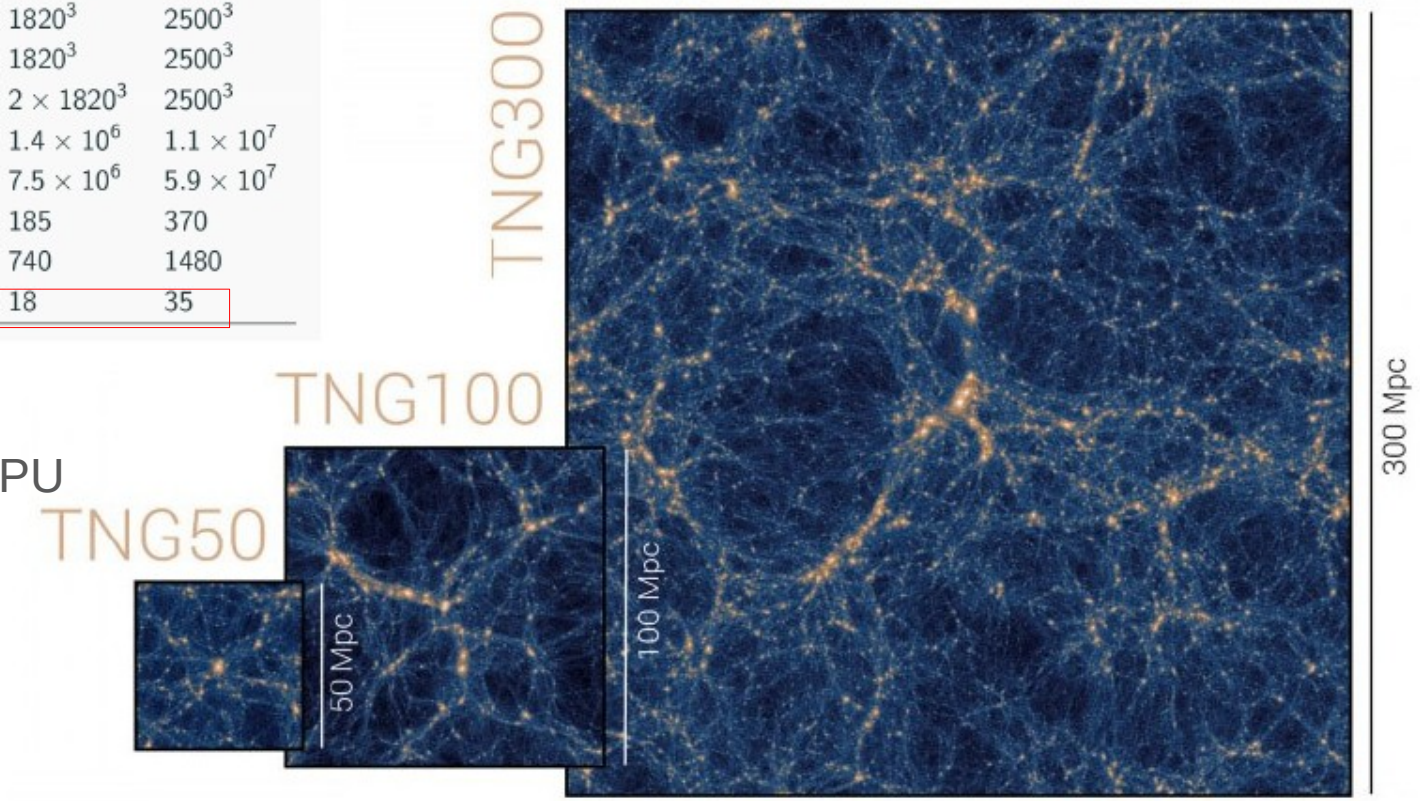


Top 500 list
Nov. 2024

Rank	System	Cores	Rmax (PFlop/s)	Rpeak (PFlop/s)	Power (kW)
1	**El Capitan** - HPE Cray EX255a, AMD 4th Gen EPYC 24C 1.8GHz, AMD Instinct MI300A, Slingshot-11, TOSS, **HPE** DOE/NNSA/LLNL **United States**	11,039,616	1,742.00	2,746.38	29,581
2	**Frontier** - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, HPE Cray OS, **HPE** DOE/SC/Oak Ridge National Laboratory **United States**	9,066,176	1,353.00	2,055.72	24,607
3	**Aurora** - HPE Cray EX - Intel Exascale Compute Blade, Xeon CPU Max 9470 52C 2.4GHz, Intel Data Center GPU Max, Slingshot-11, **Intel** DOE/SC/Argonne National Laboratory **United States**	9,264,128	1,012.00	1,980.01	38,698
4	**Eagle** - Microsoft NDv5, Xeon Platinum 8480C 48C 2GHz, NVIDIA H100, NVIDIA Infiniband NDR, **Microsoft Azure** Microsoft Azure **United States**	2,073,600	561.20	846.84	
5	**HPC6** - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, RHEL 8.9, **HPE** Eni S.p.A. **Italy**	3,143,520	477.90	606.97	8,461
6	**Supercomputer Fugaku** - Supercomputer Fugaku, A64FX 48C 2.2GHz, Tofu interconnect D, **Fujitsu** RIKEN Center for Computational Science **Japan**	7,630,848	442.01	537.21	29,899

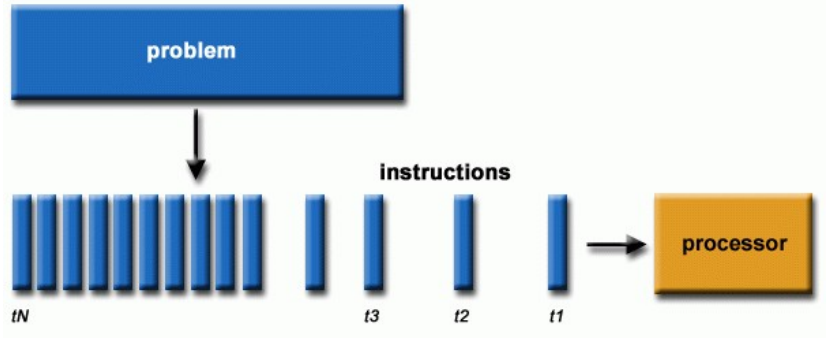
Run Name		**TNG50**	TNG100	TNG300
Volume	$[\mathrm{Mpc}^3]$	$\mathbf{51.7^3}$	110.7^3	302.6^3
L_{box}	$[\mathrm{Mpc}/h]$	**35**	75	205
N_{GAS}	-	$\mathbf{2160^3}$	1820^3	2500^3
N_{DM}	-	$\mathbf{2160^3}$	1820^3	2500^3
N_{TR}	-	$\mathbf{2160^3}$	2×1820^3	2500^3
m_{baryon}	$[\mathrm{M}_\odot]$	$\mathbf{8.5 \times 10^4}$	1.4×10^6	1.1×10^7
m_{DM}	$[\mathrm{M}_\odot]$	$\mathbf{4.5 \times 10^5}$	7.5×10^6	5.9×10^7
$\epsilon_{gas,min}$	[pc]	**74**	185	370
$\epsilon_{DM,stars}$	[pc]	**288**	740	1480
CPU Time	[Mh]	**130**	18	35

~ 15000 years on 1 CPU
For TNG50



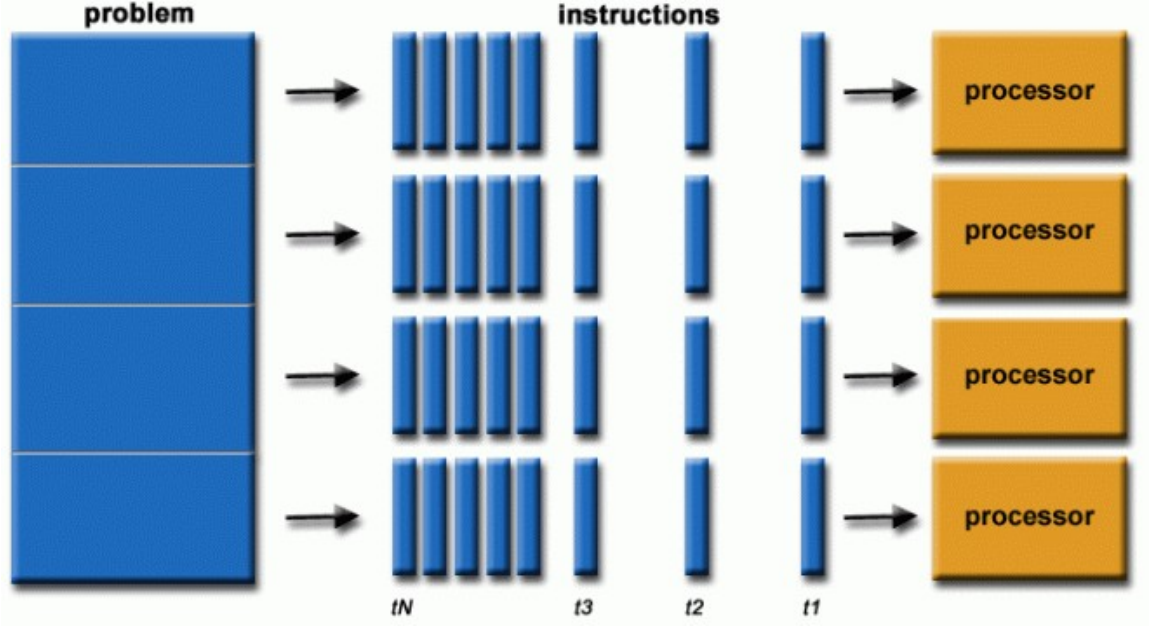
Serial Computing

- Traditionally, software has been written for serial computation:
  - A problem is broken into a discrete series of instructions
  - Instructions are executed sequentially one after another
  - Executed on a single processor
  - Only one instruction may execute at any moment in time



Parallel Computing

In the simplest sense, parallel computing is the simultaneous use of multiple compute resources to solve a computational problem:



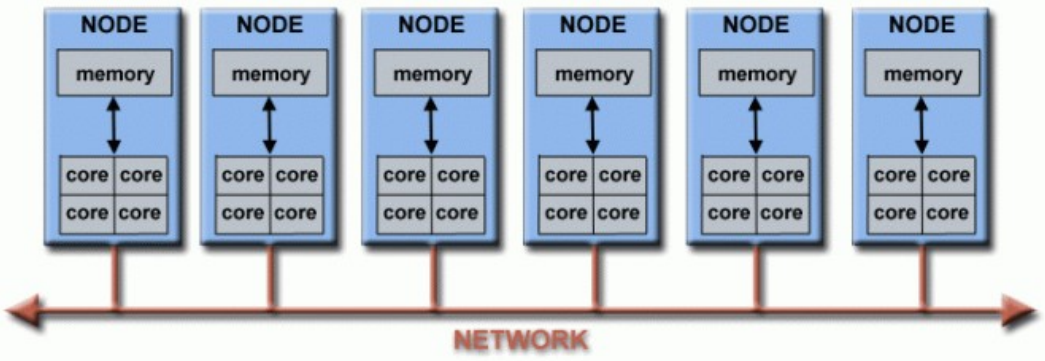
Parallel Computers

Virtually all stand-alone computers today are parallel from a **hardware** perspective:

- Multiple functional units (L1 cache, L2 cache, branch, prefetch, decode, floating-point, graphics processing (GPU), integer, etc.)
- Multiple execution units/cores
- Multiple hardware threads

Parallel Computers

Networks connect multiple stand-alone computers (nodes) to make larger parallel computer clusters.

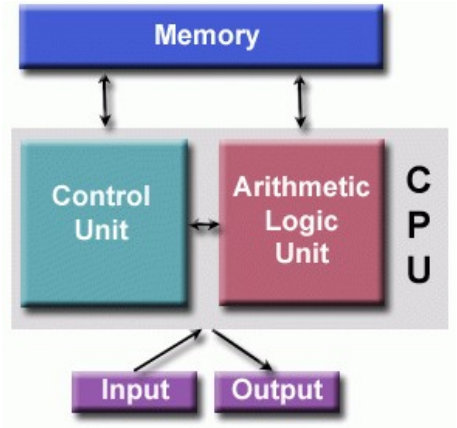


By Carlos Jones/ORNL - flickr, CC BY 2.0, https://commons.wikimedia.org/w/index.php?curid=73905694

von Neumann Computer Architecture

He first authored the general requirements for an electronic computer in his 1945 papers.

Since then, virtually all computers have followed this basic design:



General Parallel Computing Terminology

Process:

An independent instance of a running program with its own memory space, resources, and state. Each process has:

- A separate address space (memory isolation).
- Its own file descriptors, environment variables, and security context.
- High overhead for creation and management (heavyweight).
- Processes do not share memory by default; inter-process communication (IPC) mechanisms like pipes, sockets, or shared memory are required.
-

Thread:

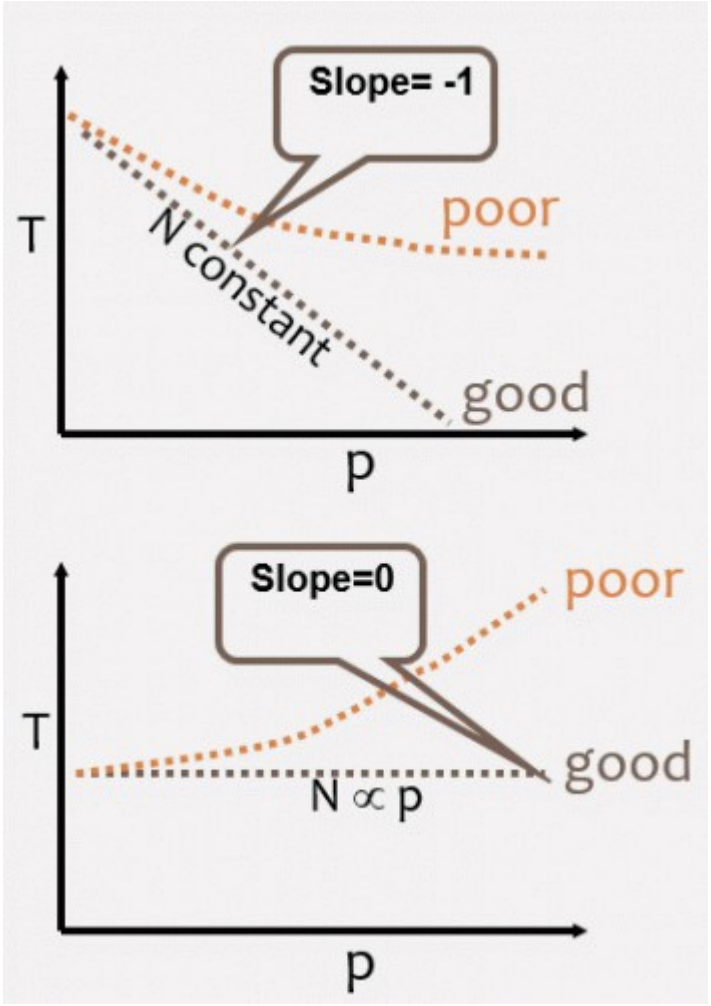
Is a lightweight unit of execution within a process, sharing the same memory space and resources. Threads within the same process:

- Share the same address space (can access global variables directly).
- Have their own stack and registers but share heap memory.Lower overhead for creation and switching (lightweight).
- Communication between threads is faster (via shared memory) but requires synchronization (e.g., mutexes, semaphores) to avoid race conditions.

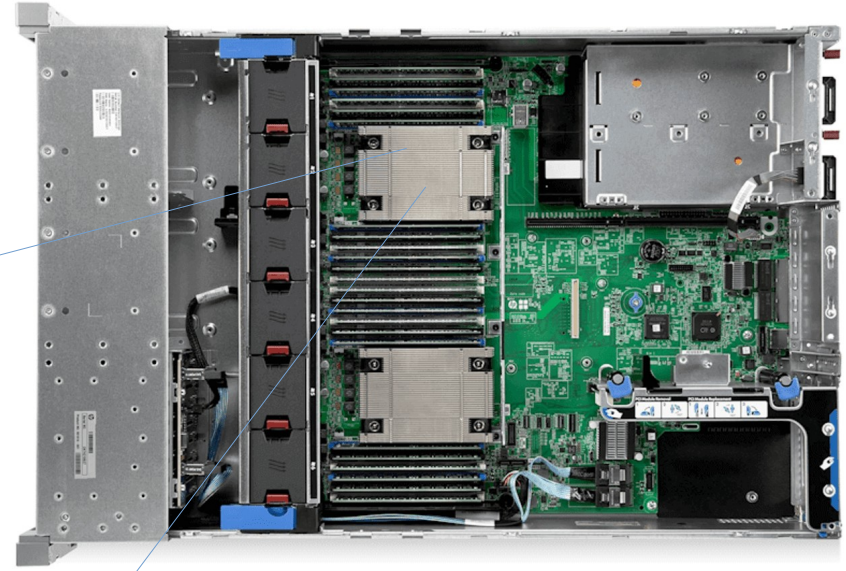
Scalability

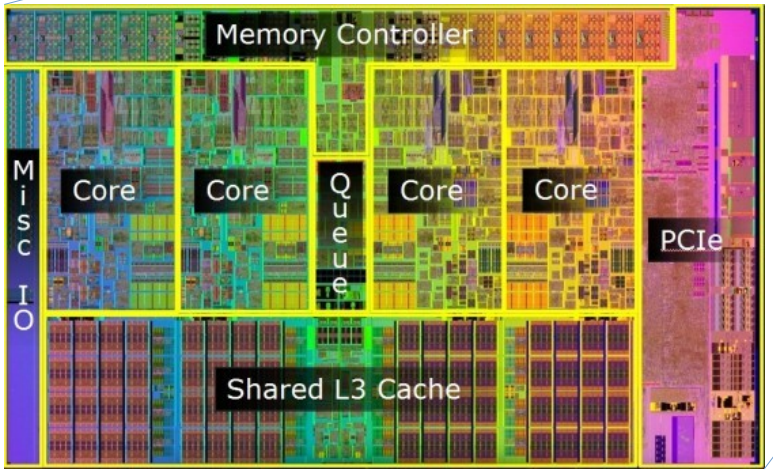
Strong scaling

Weak scaling



Typical Compute node





Typical modern CPU

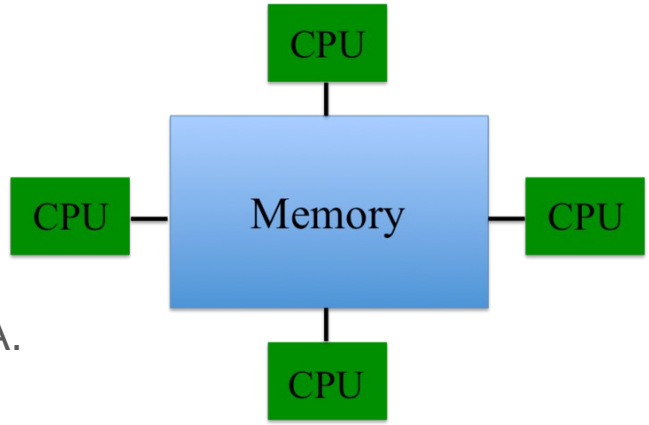
Parallel Computer Memory Architectures

- There are three main approaches to parallelization based on the memory architecture.
  - Shared memory
  - Distributed memory
  - Hybrid approach

Shared Memory

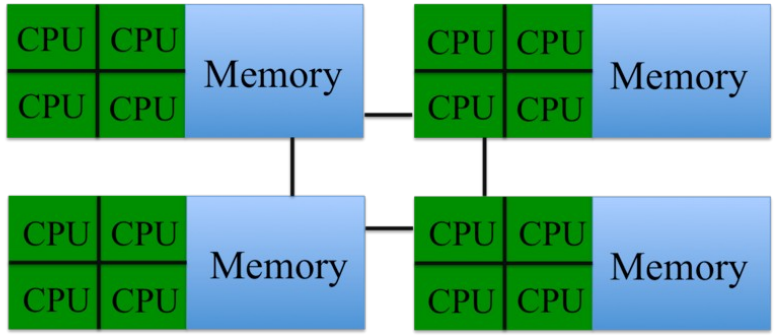
Uniform Memory Access (UMA)

- Most commonly represented today by Symmetric Multiprocessor (SMP) machines
- Identical processors
- Equal access and access times to memory
- Sometimes called CC-UMA - Cache Coherent UMA.
- Cache coherent means if one processor updates a location in shared memory, all the other processors know about the update. Cache coherency is accomplished at the hardware level.



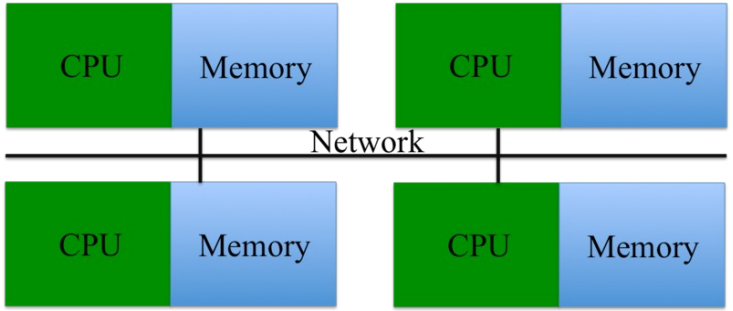
Non-Uniform Memory Access (NUMA):

- Often made by physically linking two or more SMPs
- One SMP can directly access memory of another SMP
- Not all processors have equal access time to all memories
- Memory access across link is slower
- If cache coherency is maintained, then may also be called CC-NUMA - Cache Coherent NUMA



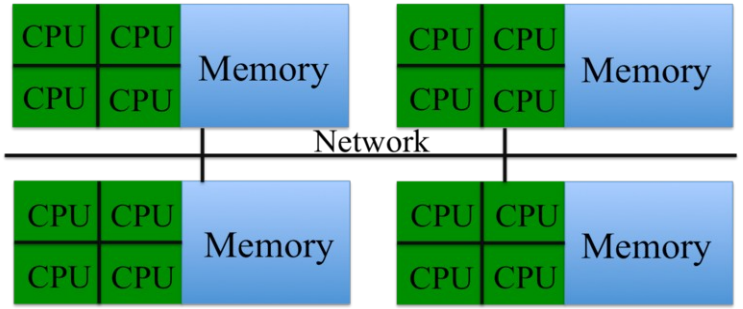
Distributed Memory

- Distributed memory systems require a **communication network** to connect inter-processor memory.
- Processors have their own **local memory**, so there is **no concept of global address** space across all processors.
- Programmer has to explicitly define how and when data is communicated.
- Synchronization between tasks is likewise the programmer's responsibility.
- The network "fabric" used for data transfer varies widely, though it can be as simple as Ethernet, or be complex as Infiniband.



Hybrid Distributed-Shared Memory

- The largest and fastest computers in the world today employ both shared and distributed memory architectures.
- The shared memory component can be a shared memory machine and/or graphics processing units (GPU).
- Still network communications are required to move data from one machine to another.
- Current trends seem to indicate that this type of memory architecture will continue to prevail and increase at the high end of computing for the foreseeable future.



Shared memory programming using threads

UNIX **processes** are isolated from each other - they have there own protected memory.

A process can however be split up into multiple execution paths, so-called **threads**, allowing lightweight parallelism where data-sharing is trivially achieved.

Threads can be created by the programmer explicitly through the pthreads system calls, or via **OpenMP**.

In OpenMP, one can easily create and destroy threads through a simple language extension.

OpenMP is an application programming interface (API) that supports multi-platform shared-memory multiprocessing programming in C, C++, and Fortran on many operating systems such as Gnu/Linux and Windows.

Directive-based (#pragma omp in C/C++ and !$omp Fortran).
Easy to implement (minimal code changes).
Supports multi-core CPUs.

Previous Official OpenMP Specifications

- OpenMP API 5.1 Specification – Nov 2020 – HTML Version – Softcover version on Amazon
- OpenMP 5.1 Examples – Aug 2021
- OpenMP 5.1 Additional Definitions – Nov 2020
- OpenMP 5.1 Reference Guide (PDF)

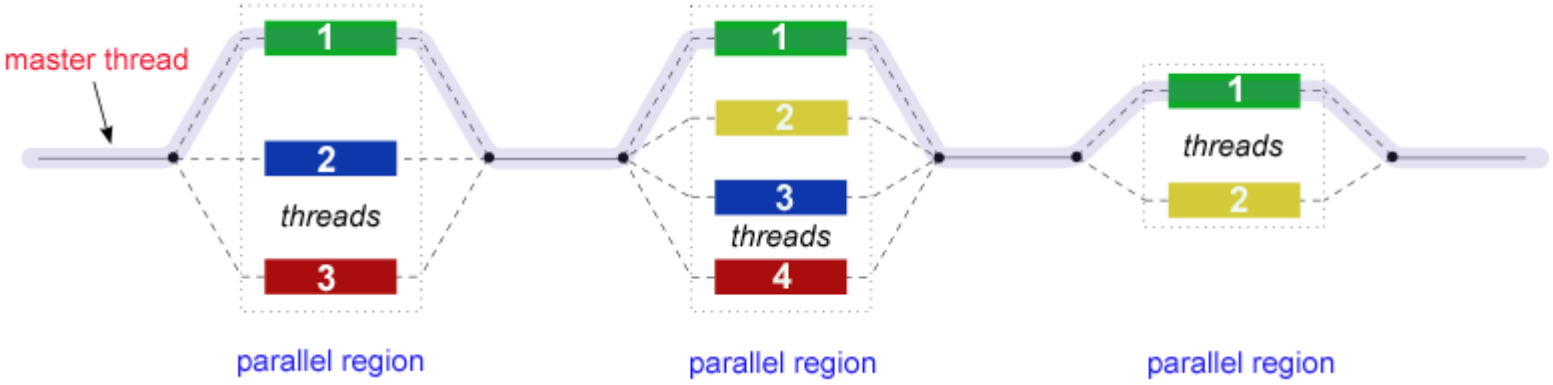
- OpenMP API 5.0 Specification – Nov 2018 – HTML Version – Softcover version on Amazon
- OpenMP API Context Definitions 1.0 – Jan 2020
- OpenMP API 5.0 Reference Guide – Japanese Translation
- OpenMP API 5.0 Supplementary Source Code
- OpenMP API 5.0.1 Examples – June 2020

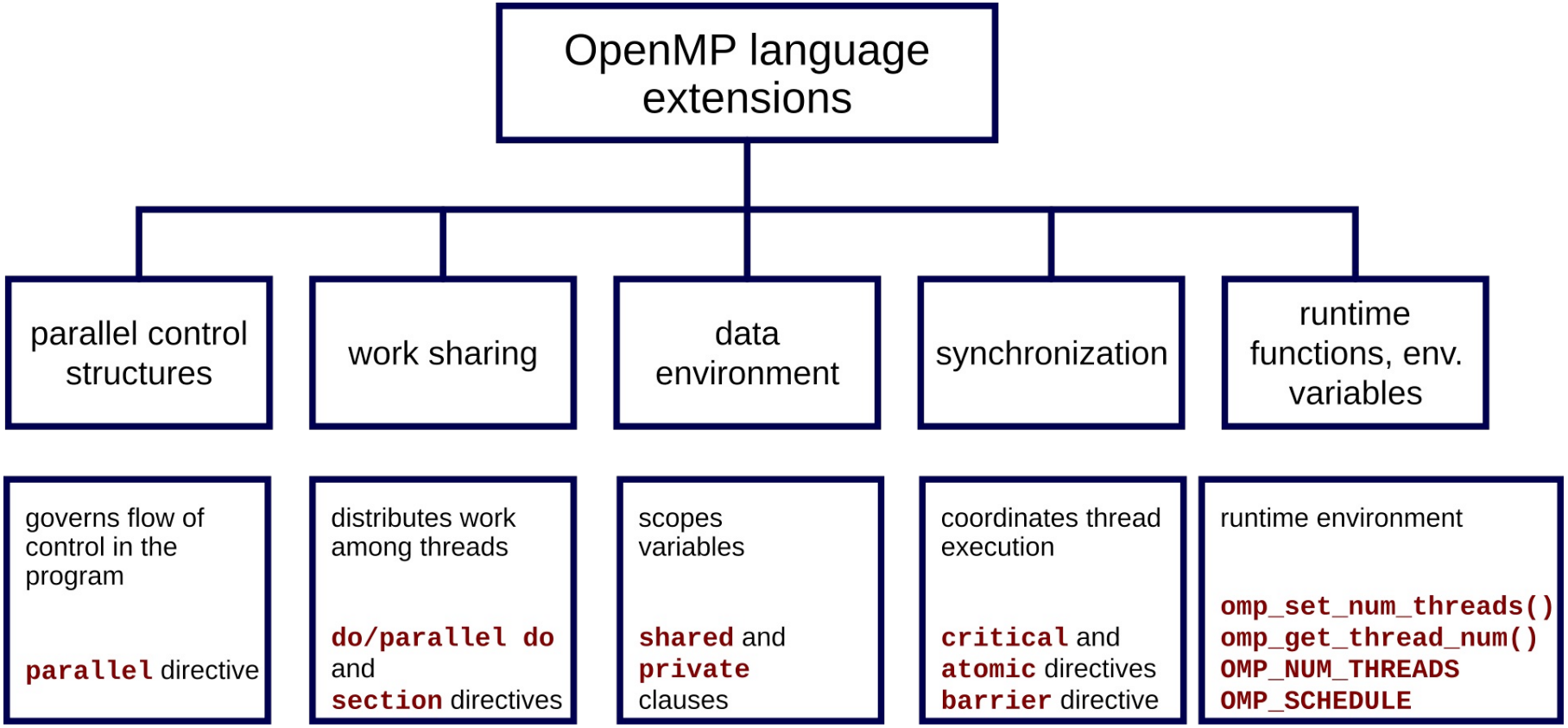
- OpenMP 4.5 Complete Specifications – Nov 2015
- OpenMP 4.5 Reference Guide – C/C++ – Nov 2015
- OpenMP 4.5 Reference Guide – Fortran – Nov 2015
- OpenMP 4.5 Examples (Nov 2016) *pdf*

- OpenMP 4.0 Complete Specifications – Jul 2013
- OpenMP 4.0 Reference Guide – C/C++ – October 2013
- OpenMP 4.0 Reference Guide – Fortran – October 2013
- OpenMP Examples 4.0.2 – Mar 2015
- OpenMP 4.0.1 Examples – Feb 2014

- Version 3.1 Complete Specifications – Jul 2011
- Version 3.1 Summary Card C/C++ – Sep 2011
- Version 3.1 Summary Card Fortran – Sep
- Version 3.0 Complete Specifications – May 2008
- Version 3.0 Summary Card C/C++ – Nove 2008
- Version 3.0 Summary Card Fortran – Revised Mar 2009

Fork-Join model in OpenMP




OpenMP language extensions
parallel control structures
work sharing
data environment
synchronization
runtime functions, env. variables
governs flow of control in the program
parallel directive
distributes work among threads
do/parallel do and section directives
scopes variables
shared and private clauses
coordinates thread execution
critical and atomic directives
barrier directive
runtime environment
omp_set_num_threads()
omp_get_thread_num()
OMP_NUM_THREADS
OMP_SCHEDULE

Example:

```c
#include <stdio.h>
#include <omp.h>

int main() {

    // Get and print max available threads
    int max_threads = omp_get_max_threads();
    printf("Number of available threads: %d\n\n", max_threads);

    // Parallel region
    #pragma omp parallel
    {
        int thread_id = omp_get_thread_num();
        printf("Thread %d says: Hello!\n", thread_id);
    }

    return 0;
}
```

```fortran
program openmp_demo
    use omp_lib             ! Required for OpenMP functionality
    implicit none

    integer :: max_threads  ! Stores maximum available threads
    integer :: thread_id    ! Stores individual thread identifier

    ! Query and display available threads
    max_threads = omp_get_max_threads()
    print '(A,I0)', 'Available threads: ', max_threads

    ! Parallel execution section
    !$omp parallel private(thread_id)
        thread_id = omp_get_thread_num()
        print '(A,I0,A)', 'Thread ', thread_id, ' active'
    !$omp end parallel

end program openmp_demo
```

To compile:

```
gfortran parallel_hello_world.f90 -o parallel_hello_world.exe -fopenmp
ifort parallel_hello_world.f90 -o parallel_hello_world.exe -qopenmp
```

Important environment variable: OMP_NUM_THREADS

When parallelizing code with OpenMP, consider these key aspects to ensure **correctness**, **efficiency**, and **scalability**:

1. Understanding Work Distribution

- Identify loops and tasks that can be parallelized.
- Ensure workload is evenly distributed among threads to avoid bottlenecks.

2. Managing Shared vs. Private Variables

- Use private, shared, and reduction clauses appropriately to prevent race conditions.
- Avoid unnecessary sharing of variables to minimize synchronization overhead.

3. Synchronization and Race Conditions

- Protect shared resources using critical, atomic, or ordered constructs.
- Minimize synchronization points to reduce performance degradation.

4. Load Balancing

- Use scheduling strategies (static, dynamic, guided) to optimize thread workload distribution.
- Consider loop iteration granularity to achieve better load balancing.

6. Thread Overhead and Scalability

- Avoid excessive thread creation; balance computation vs. overhead.

7. Nested Parallelism & Task Parallelism

- Determine if enabling nested parallel regions (OMP_NESTED) benefits your application.
- Use OpenMP tasks (#pragma omp task) for irregular workloads

8. Debugging and Performance Analysis

- Utilize OpenMP environment variables like OMP_NUM_THREADS to tune performance.
- Profile execution time with tools like GNU gprof and TAU.

OpenMP pragma, function, or clause	Concepts
#pragma omp parallel	Parallel region, teams of threads, structured block, interleaved execution across threads.
void omp_set_thread_num() int omp_get_thread_num() int omp_get_num_threads()	Default number of threads and internal control variables. SPMD pattern: Create threads with a parallel region and split up the work using the number of threads and the thread ID.
double omp_get_wtime()	Speedup and Amdahl's law. False sharing and other performance issues.
setenv OMP_NUM_THREADS N	Setting the internal control variable for the default number of threads with an environment variable
#pragma omp barrier #pragma omp critical	Synchronization and race conditions. Revisit interleaved execution.
#pragma omp for #pragma omp parallel for	Worksharing, parallel loops, loop carried dependencies.
reduction(op:list)	Reductions of values across a team of threads.
schedule (static [,chunk]) schedule(dynamic [,chunk])	Loop schedules, loop overheads, and load balance.
shared(list), private(list), firstprivate(list)	Data environment.
default(none)	Force explicit definition of each variable’s storage attribute
nowait	Disabling implied barriers on workshare constructs, the high cost of barriers, and the flush concept (but not the flush directive).
#pragma omp single	Workshare with a single thread.
#pragma omp task #pragma omp taskwait	Tasks including the data environment for tasks.

Other implementations:

Microsoft threads

Java threads

Python threads

CUDA threads for GPU

Python libraries that release GIL:

Numba: loop parallelization with prange

PyOMP: OpenMP in Numba

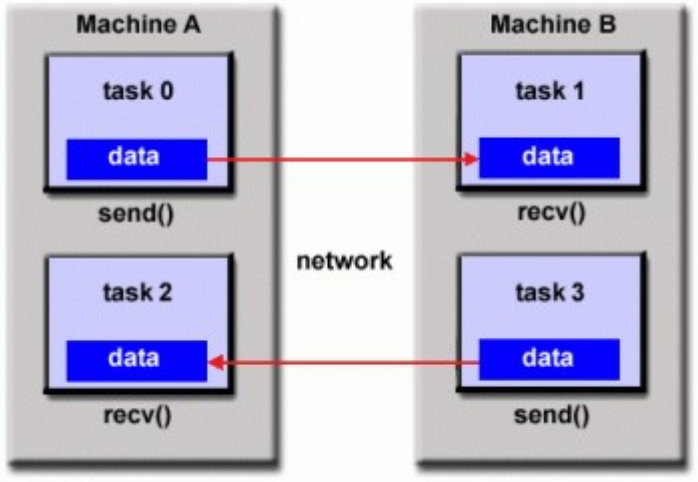
```
from numba import njit
from numba.openmp import openmp_context as openmp
from numba.openmp import omp_get_wtime

@njit
def piFunc(NumSteps):
    step = 1.0/NumSteps
    sum = 0.0
    startTime = omp_get_wtime()

    with openmp ("parallel for private(x) reduction(+:sum)"):
        for i in range(NumSteps):
            x = (i+0.5)*step
            sum += 4.0/(1.0 + x*x)

    pi = step*sum
    runTime = omp_get_wtime() - startTime
    print(pi, NumSteps, runTime)
    return pi

start = omp_get_wtime()
pi = piFunc(100000000)
print("time including JIT time",omp_get_wtime()-start)
```

Distributed memory / Message Passing Model

A set of tasks that use their own local memory during computation. Multiple tasks can reside on the same physical machine and/or across an arbitrary number of machines.

Tasks exchange data through communications by sending and receiving messages.

Data transfer usually requires cooperative operations to be performed by each process. For example, a send operation must have a matching receive operation.



MPI Implementations

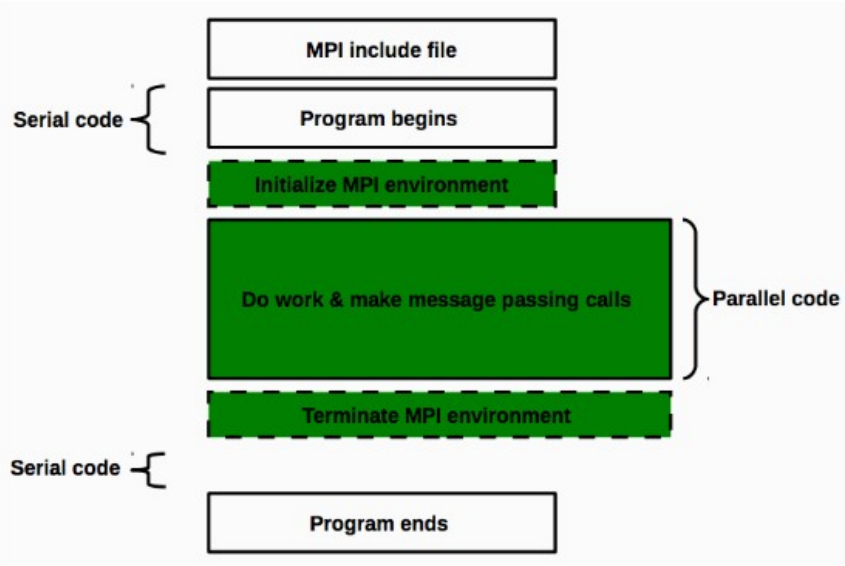
Historically, a variety of message passing libraries have been available since the 1980s. These implementations differed substantially from each other making it difficult for programmers to develop portable applications.

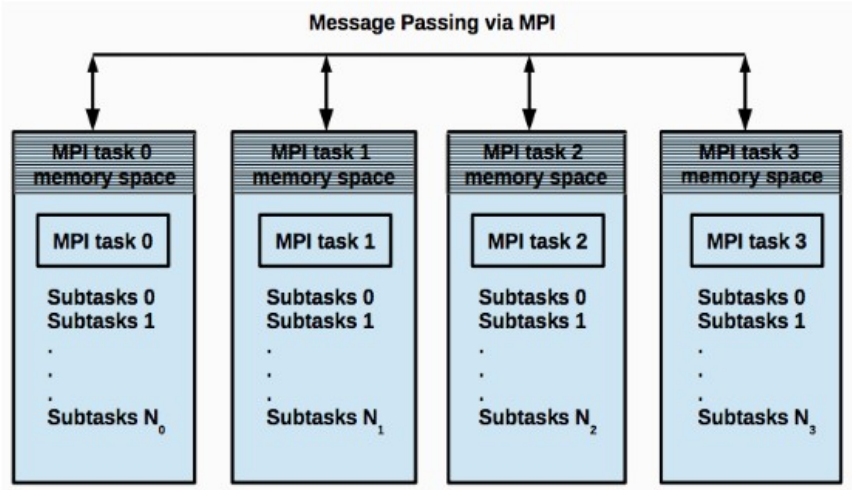
In 1992, the MPI Forum was formed with the primary goal of establishing a standard interface for message passing implementations.

Part 1 of the Message Passing Interface (MPI) was released in 1994. Part 2 (MPI-2) was released in 1996, MPI-3 in 2012 and MPI-4 in 2021.

MPI is the "de facto" industry standard for message passing, replacing virtually all other message passing implementations used for production work.

General structure of MPI programming model





Data decomposition and all communication in the parallel code needs to be coded explicitly.

Doing this well, represents a steep learning curve.

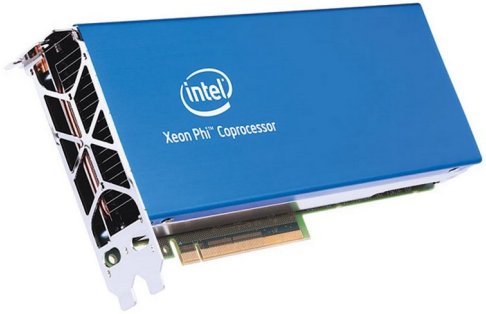
scalability issue

- MPI is a well-established and portable standard, yet modern MPI libraries encounter scalability issues due to high memory demands, particularly in all-to-all communication scenarios.
- One solution: Hybrid parallelism (MPI + OpenMP + GPU acceleration).
- Combining MPI with shared-memory parallelism and GPU acceleration is helping mitigate memory bottlenecks in large-scale simulations.

Vector extensions

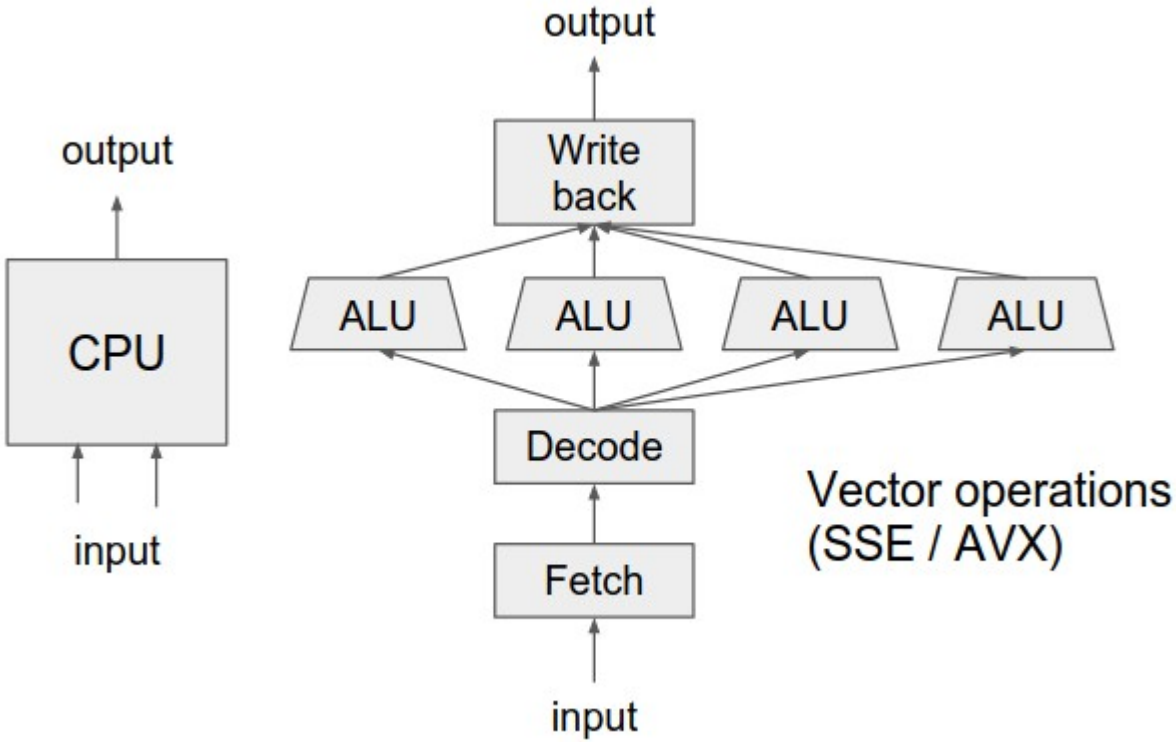
- AVX, introduced in Intel Sandy bridge, offers registers that are 256 bit wide.
- With those, we can run:
  - 4 double precision calculations in parallel, or
  - 8 single precision calculations in parallel

  in roughly the same time as a single operation.



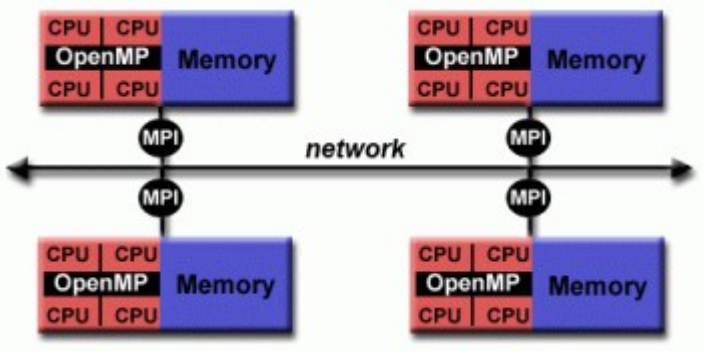
- In the Intel Xeon Phi Accelerator Cards, the width of these vector instructions has been doubled yet again to 512 bit.

Vector extensions



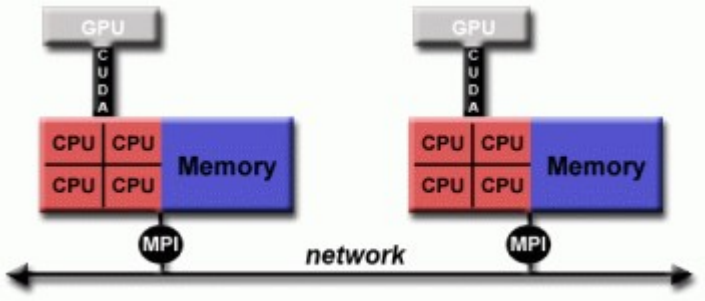
https://dlsys.cs.washington.edu/pdf/lecture5.pdf

Hybrid Model: MPI + OpenMP



- A hybrid model combines more than one of the previously described programming models.
- Currently, a common example of a hybrid model is the combination of the MPI with the threads model (OpenMP).
- Threads perform computationally intensive kernels using local, on-node data.
- Communications between processes on different nodes occurs over the network using MPI.

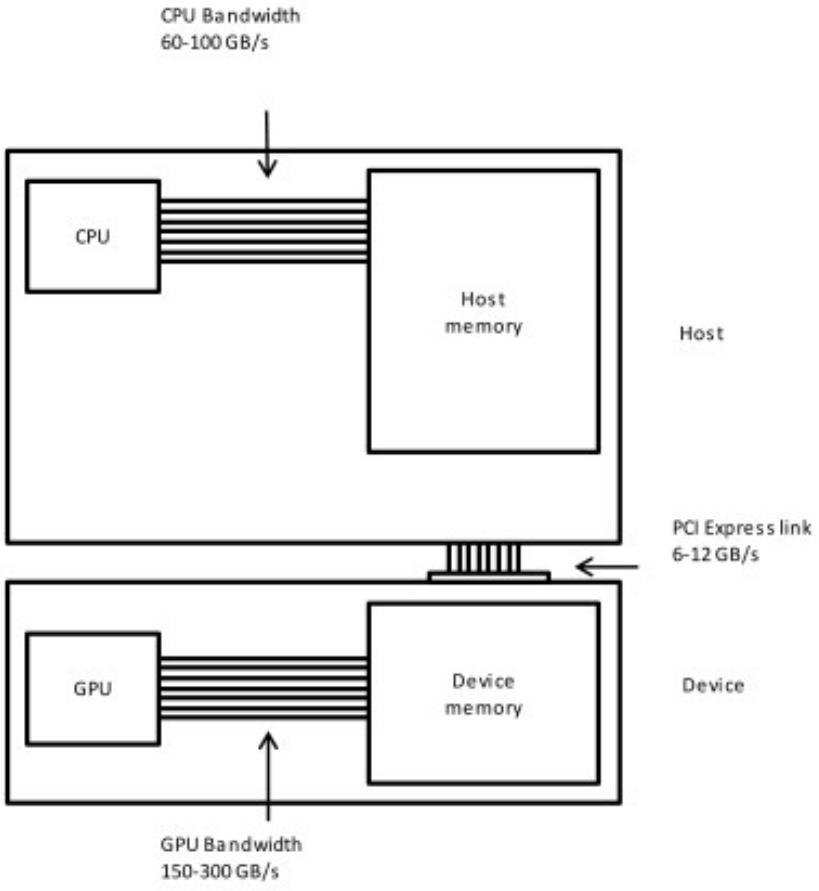
Hybrid Model: MPI + GPU



- MPI tasks run on CPUs using local memory and communicating with each other over a network.

- Computationally intensive kernels are off-loaded to GPUs on-node.
- Data exchange between node-local memory and GPUs uses CUDA (or something equivalent).

GPU - Graphical Processing Unit



GPU

Advantage:

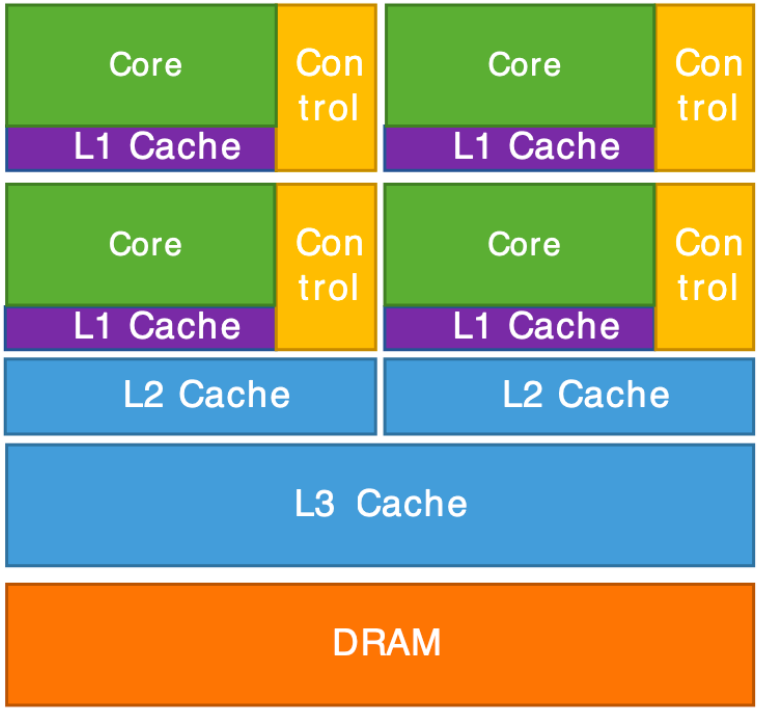
- A highly specialized unit designed to maximize simultaneous calculations, enabling massive parallelism.

Key Limitations:

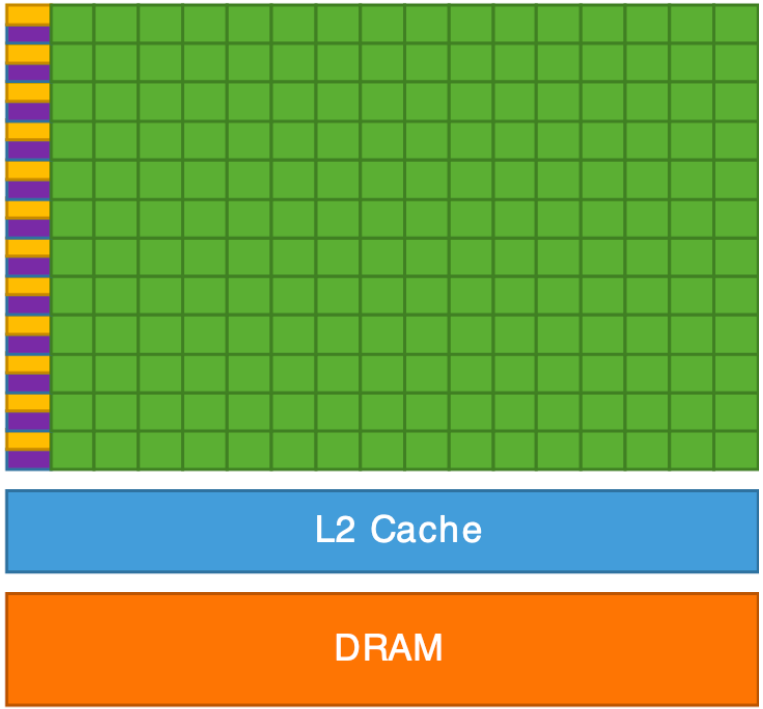
- Programs must be explicitly optimized (e.g., using CUDA, OpenCL, or ROCm) to leverage GPU acceleration

Astrophysics is not so simple!

Simple comparison of CPU and GPU



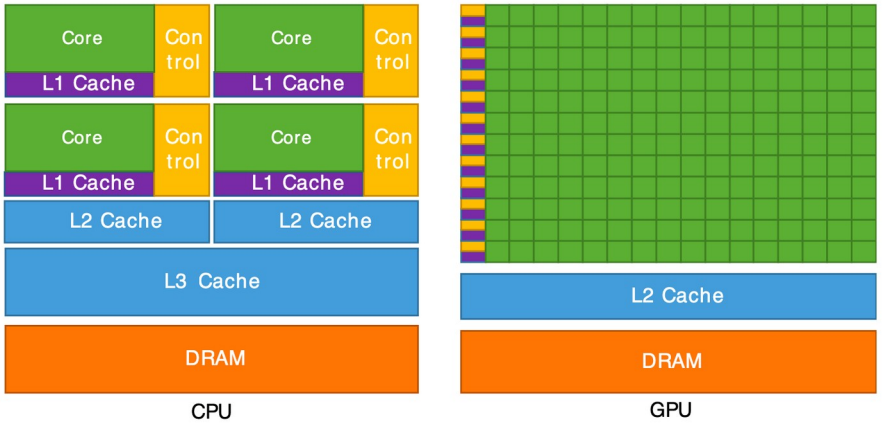
CPU



GPU

Simple comparison of CPU and GPU

- CPUs can handle more complex workflows compared to GPUs.
- CPUs don't have as many arithmetic logic units or floating point units as GPUs, but the ALUs and FPUs in a CPU core are individually more capable.
- CPUs have more cache memory than GPUs.



- GPUs are really designed for workloads that **can be parallelized to a significant degree**. This is indicated in the diagram by having just one gold control box for every row of the little green computational boxes.

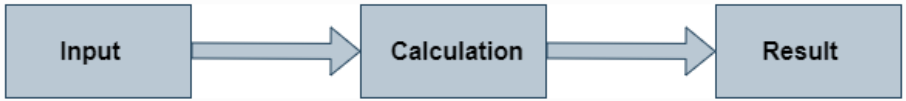
How to write programs on GPU?

- **CUDA** (Compute Unified Device Architecture): Released by NVIDIA in 2007 and only works for its GPU. Works with C, C++, Fortran, Python, Julia, Matlab, etc.
- **ROCm** (Radeon Open Compute): Released by AMD in 2016. Not mature as CUDA.
- **OpenCL** (Open Computing Language): Released by Khronos group in 2009. For Cross-platform GPU/CPU/accelerator computing. Steeper learning curve.
- **HIP** (Heterogeneous-computing Interface for Portability): A C++ API to write portable GPU code that runs on both AMD (via ROCm) and NVIDIA (via CUDA). Developed by AMD in 2016. Ports CUDA code to AMD/NVIDIA GPUs with minimal changes.
- **OpenACC** (open accelerators): Released on 2012. Can be used via NVIDIA HPC SDK .

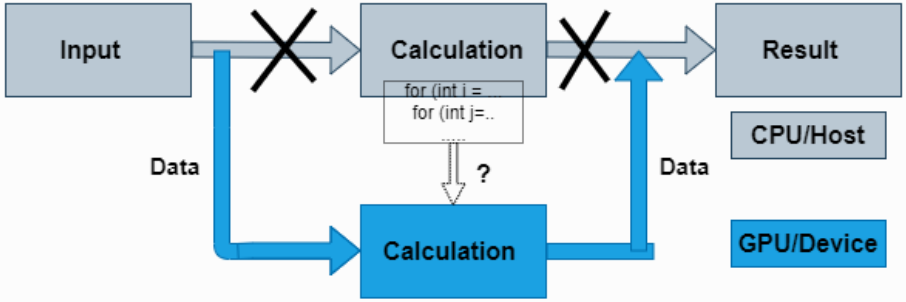
OpenACC

More Science, Less Programming

- Serial Computing on CPU



- Porting to GPU




```
#pragma acc data copy(A) create(Anew)
while ( error > tol  &&  iter  <  iter_max )  {
  error = 0.0;
#pragma acc kernels
  {
#pragma acc loop independent collapse(2) reduction(max:error)
  for (  int  j = 1; j < n-1;  j++ )  {
    for (  int  i = 1; i < m-1; i++ )  {
       Anew [j] [i] = 0.25 * ( A [j] [i+1] + A [j] [i-1] +
                                          A [j-1] [i] + A [j+1] [i]);
       error = max ( error, fabs (Anew [j] [i] - A [j] [i]));
      }
    }
    ...
  }
}
```

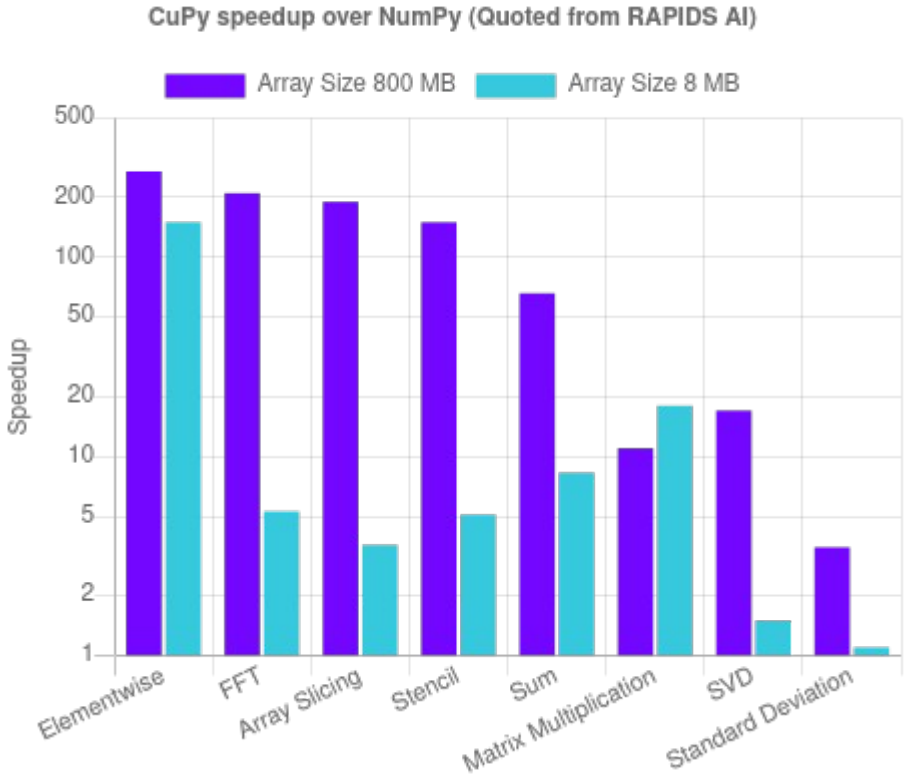

```python
from numba import cuda
import numpy
import math

# CUDA kernel
@cuda.jit
def my_kernel(io_array):
    pos = cuda.grid(1)
    if pos < io_array.size:
        io_array[pos] *= 2 # do the computation

# Host code
data = numpy.ones(256)
threadsperblock = 256
blockspergrid = math.ceil(data.shape[0] / threadsperblock)
my_kernel[blockspergrid, threadsperblock](data)
print(data)
```

```python
from numba import njit
import random

@njit
def monte_carlo_pi(nsamples):
    acc = 0
    for i in range(nsamples):
        x = random.random()
        y = random.random()
        if (x ** 2 + y ** 2) < 1.0:
            acc += 1
    return 4.0 * acc / nsamples
```

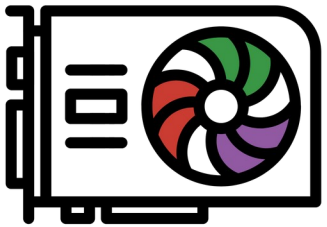
CUDA Python interfaces

CuPy

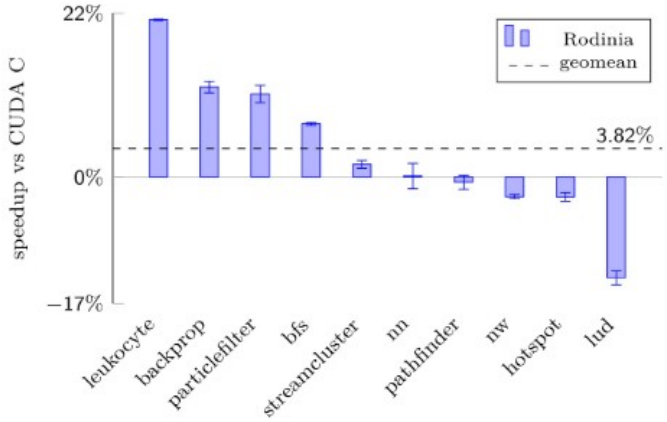
```
>>> import cupy as cp
>>> x = cp.arange(6).reshape(2, 3).astype('f')
>>> x
array([[ 0.,  1.,  2.],
       [ 3.,  4.,  5.]], dtype=float32)
>>> x.sum(axis=1)
array([  3.,  12.], dtype=float32)
```



CUDA Julia interfaces



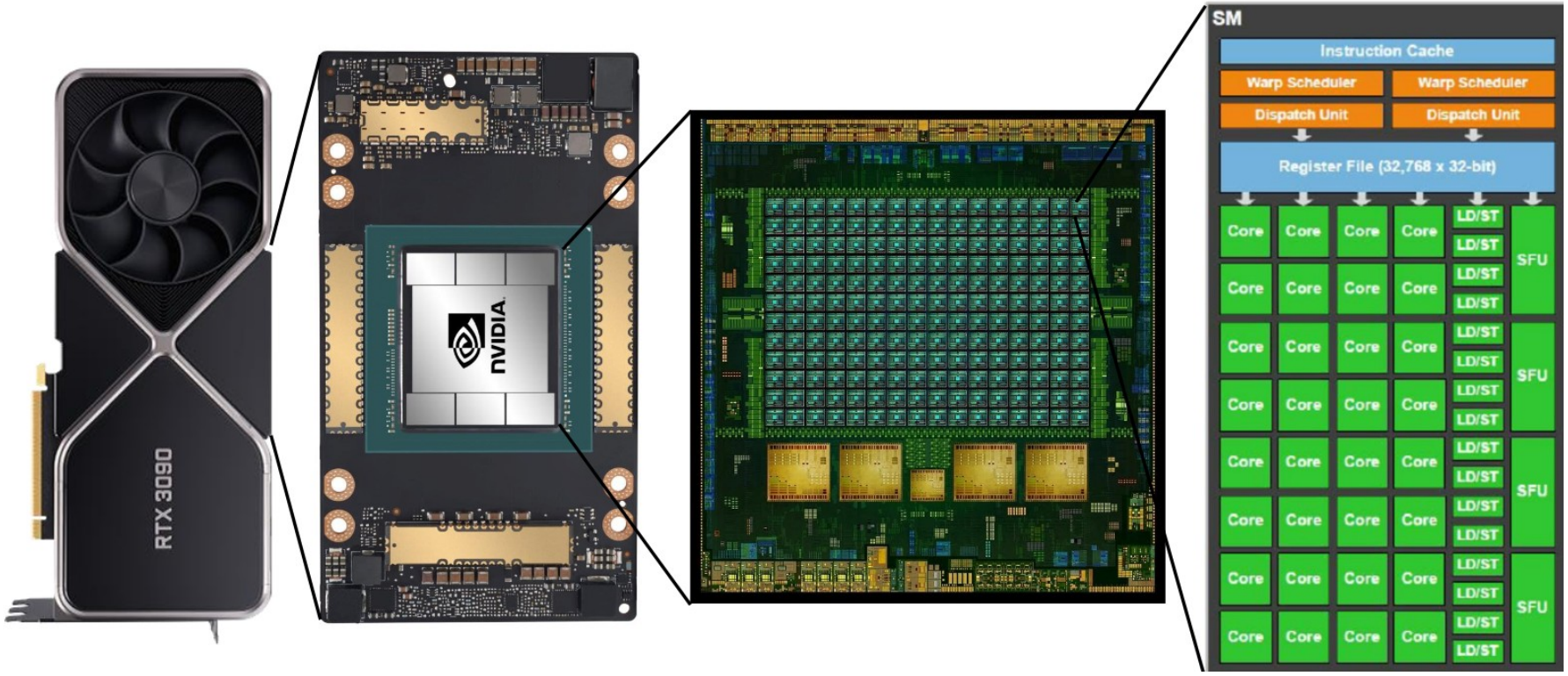
```
using BenchmarkTools
using CUDA

A = rand(2^9, 2^9);
A_d = CuArray(A);

@btime $A * $A;
# need to synchronize to let the CPU wait for the GPU kernel to finish
@btime CUDA.@sync $A_d * $A_d;
```



Performance of the Rodinia benchmark suite implemented in Julia, compared to CUDA C.

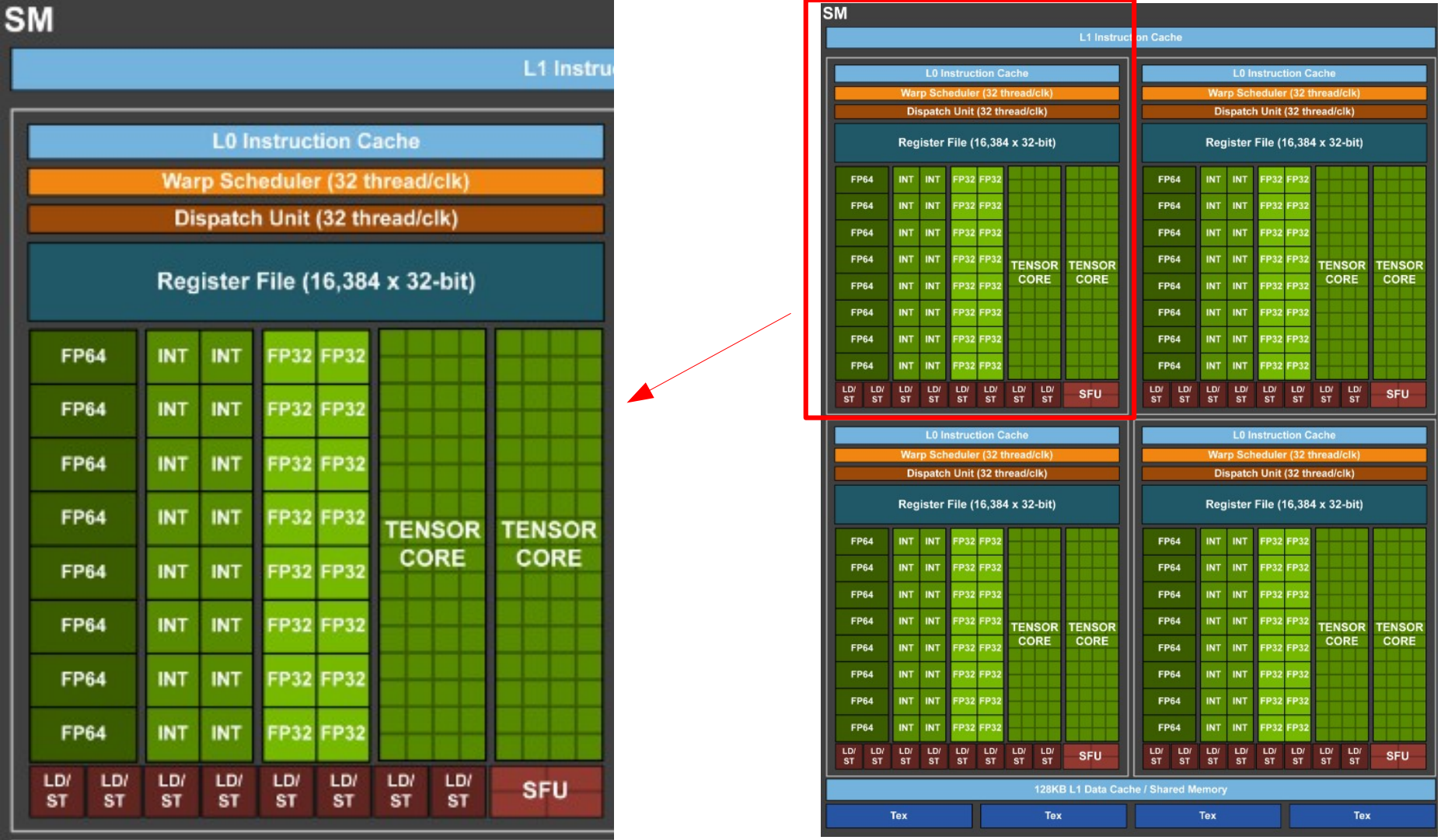
Effective Extensible Programming: Unleashing Julia on GPUs (10.1109/TPDS.2018.2872064)

GPU architecture



GPU term	Quick definition for a GPU	CPU equivalent
thread	The stream of instructions and data that is assigned to one CUDA core; note, a Single Instruction applies to Multiple Threads, acting on multiple data (SIMT)	N/A
CUDA core	Unit that processes one data item after another, to execute its portion of a SIMT instruction stream	vector lane
warp	Group of 32 threads that executes the same stream of instructions together, on different data	vector
kernel	Function that runs on the device; a kernel may be subdivided into *thread blocks*	***thread(s)***
SM, streaming multiprocessor	Unit capable of executing a thread block of a kernel; multiple SMs may work together on a kernel	***core***

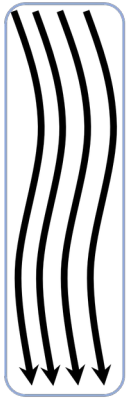
Comparison of terminology between GPUs and CPUs.


SM
L1 Instruction Cache
L0 Instruction Cache
Warp Scheduler (32 thread/clk)
Dispatch Unit (32 thread/clk)
Register File (16,384 x 32-bit)
FP64
INT
FP32
TENSOR CORE
LD/ ST
SFU
128KB L1 Data Cache / Shared Memory
Tex

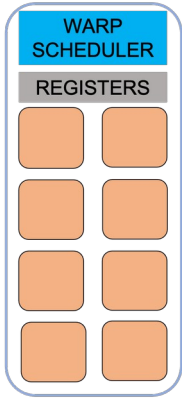
Software Hardware

CUDA THREAD

CUDA CORE

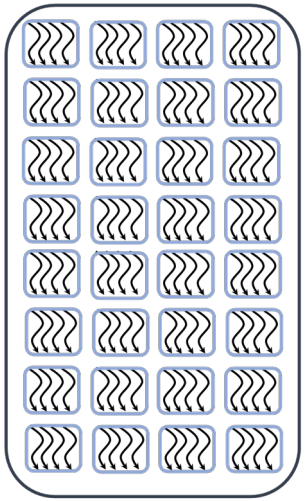


WARP

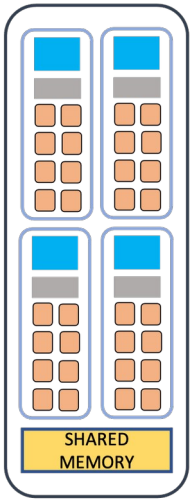


SIMT UNIT

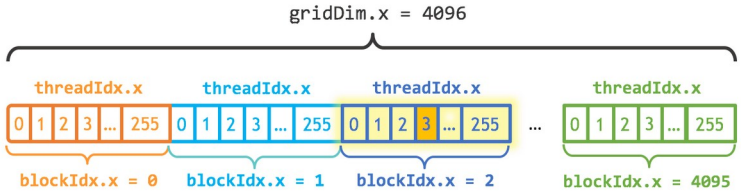
Software Hardware



BLOCK

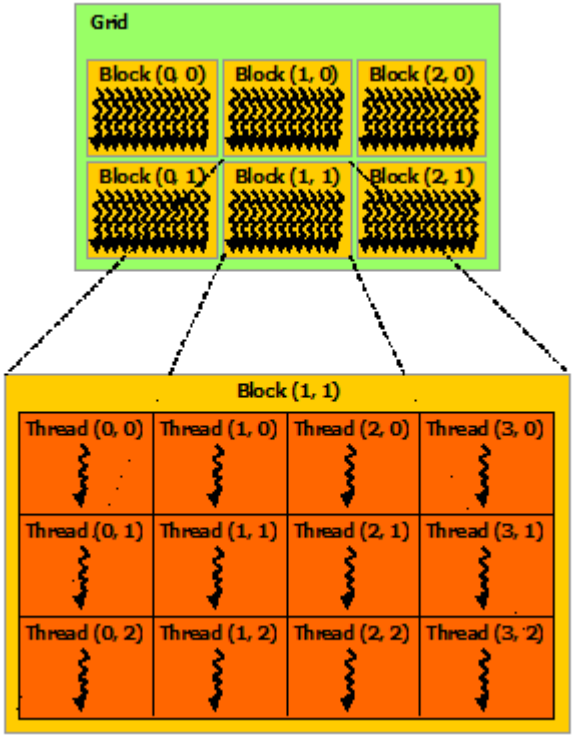


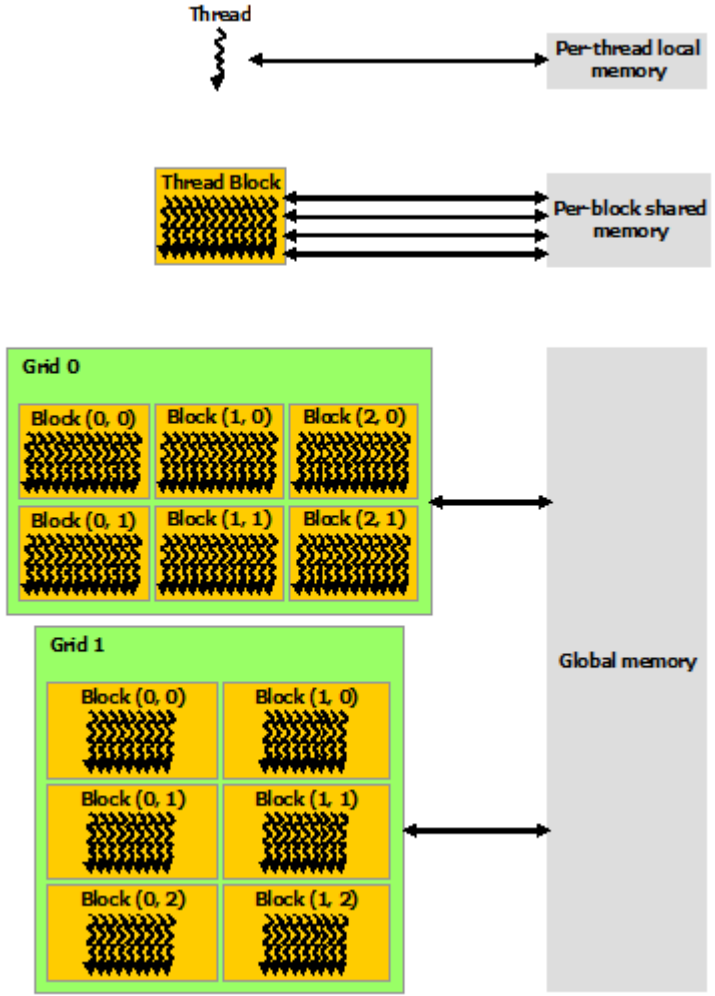
SMP




```
index = blockIdx.x  * blockDim.x + threadIdx.x
index =     (2)     *    (256)    +     (3)      = 515
```

Thread Hierarchy





Limitations for astrophysical problems

- Many astrophysics problems (e.g., N-body simulations with adaptive mesh refinement, complex gravity calculations) involve irregular data dependencies and sequential logic, which GPUs handle poorly compared to CPUs.
- Tree-based algorithms (e.g., Barnes-Hut) require frequent branching and communication, reducing GPU efficiency.
- Limited VRAM (e.g., 24–80 GB on high-end GPUs) restricts problem sizes compared to CPU clusters with distributed memory.
- High-latency data transfers between CPU and GPU can negate performance gains in multi-step simulations.

Limitations for astrophysical problems

- Double-Precision (FP64) Performance Limitations
- Many astrophysical simulations (e.g., cosmology, magnetohydrodynamics) require high-precision FP64 math, but consumer GPUs (e.g., NVIDIA GeForce) are optimized for FP32/FP16 (AI/graphics).
- Scientific GPUs (e.g., NVIDIA A100/H100, AMD MI300X) improve FP64 performance but at higher cost.
- Hybrid CPU-GPU programming (e.g., MPI + CUDA) adds complexity, requiring specialized expertise.

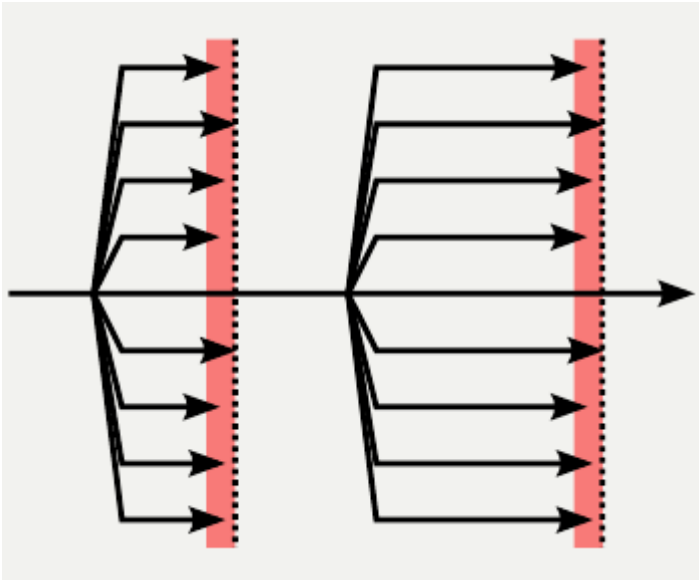
Limitations for astrophysical problems

- Scaling and memory management are major shortcoming of current astrophysical codes.
- Current scaling to 10^3 - 10^4 cores (at best).
  - Load balancing limitations
  - Significant memory overhead
  - Limited (if any) utilization of GPUs

Significant Challenges for the Next Generation:

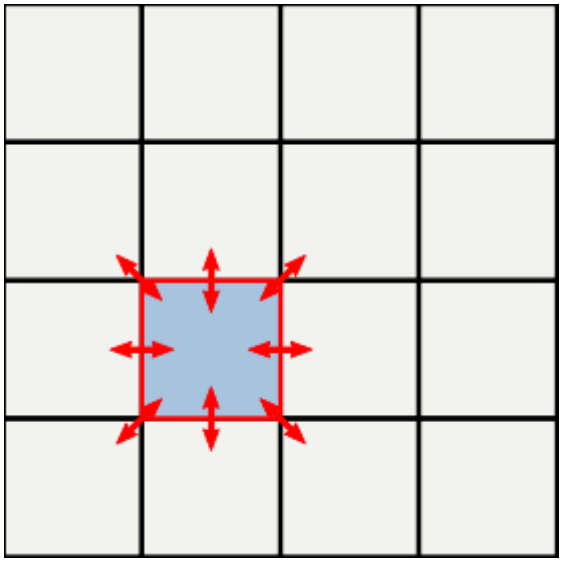
- Both MPI and OpenMP rely on the SPMD (Single Program/Multiple Data) programming model, i.e. the same bit of code is executed by all threads/nodes at more or less the same time.

- This means introducing Bottleneck to the code. **Task-based parallelization** could be the solution.



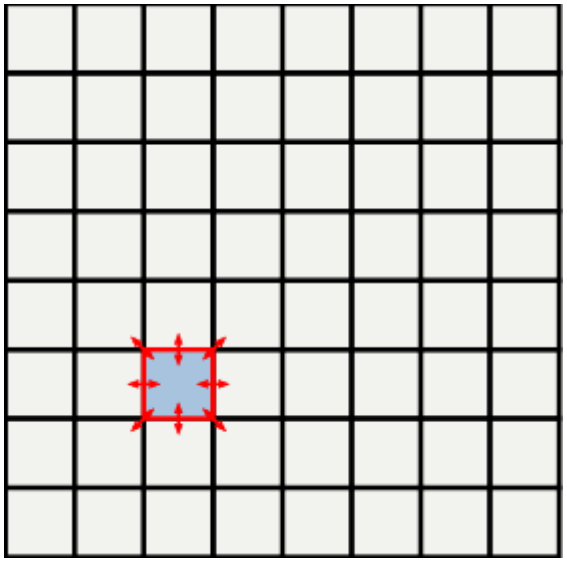
Significant Challenges for the Next Generation:

- Surface-to-volume ratio problem



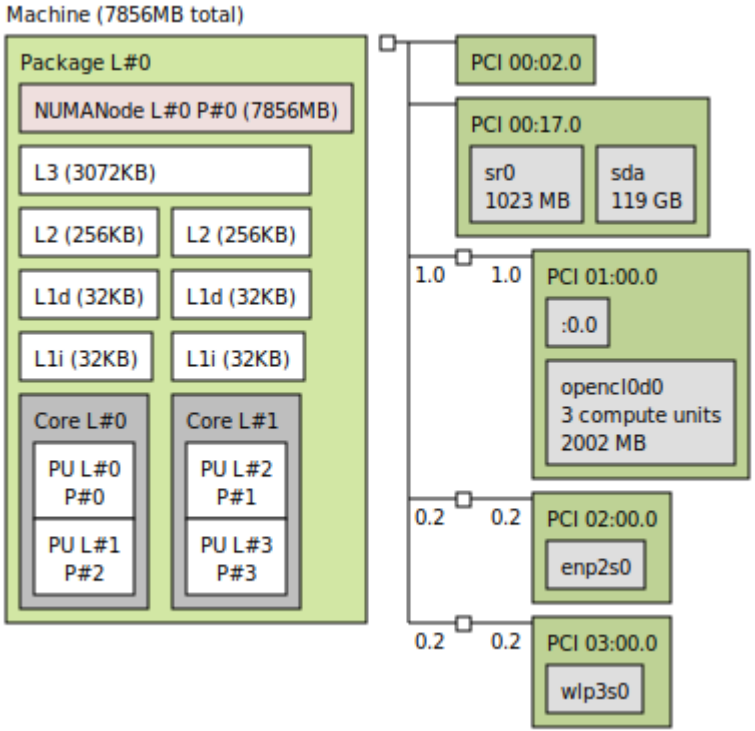
Significant Challenges for the Next Generation:

- Surface-to-volume ratio problem:
  - As the number of cores increases, the amount of computation per core (volume) decreases while the relative amount of communication (surface) increases, eventually dominating the entire computation. We can always do larger simulations, but not smaller simulations faster.

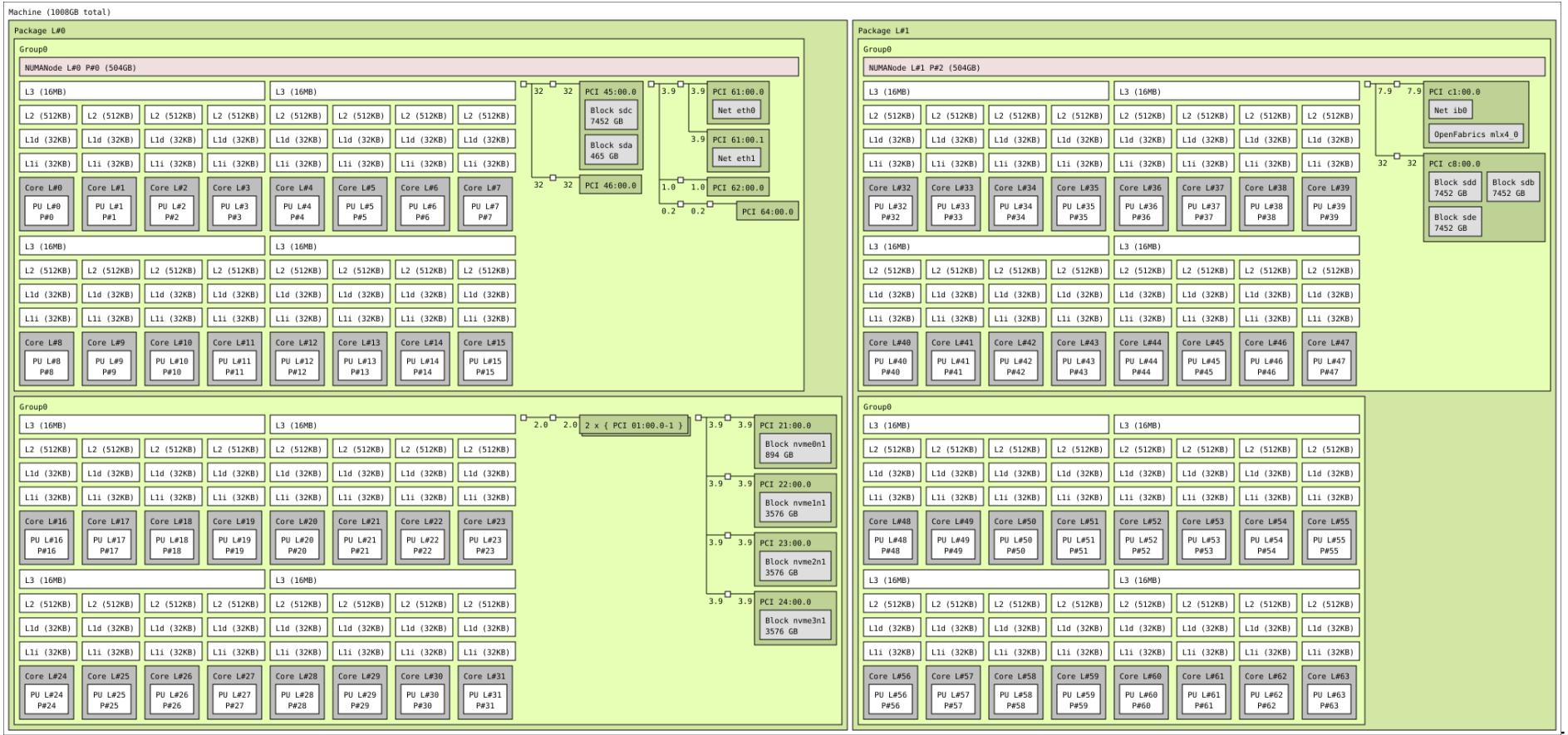


Explore the hardware you have access to

The topology of a Corei5 laptop with a Gnu/Linux system



The topology of a AMD EPYC 7502 32-Core Processor, on Sheikh Bahai national HPC center



A brief comparison of the workshop codes

	Fargo3D	Pencil	Pluto	Ramses
Type	FD	FV	FV	FV
Parallelization	MPI + CUDA	MPI	MPI + OpenACC	MPI + CUDA
Solver	(M)HD	MHD	MHD	(R)MHD
Some or main application	PPD	B evolution, dynamo, supernova	Disk, jet and star formation	Large-scale, galaxy, star cluster formation , ...
Cosmology	-	-	-	+

Awesome Astrophysical Simulation Codes

- https://github.com/pmocz/awesome-astrophysical-simulation-codes