High performance computing and GPUs

Lukas Einkemmer
University of Innsbruck

Dobbiaco summer school, 2022

Link to slides: `http://www.einkemmer.net/training.html`

Introduction

Understanding hardware is important to understand performance.

Introduction

Understanding hardware is important to understand performance.

- Make a program finish sooner/ability to solve larger problems.
- Influences how we think about and design numerical algorithms.
- Understand the performance of an algorithm before we start implementing it.
- Judge how optimized our implementation is.

Should we not have considered this all along?

Introduction

Understanding hardware is important to understand performance.

Should we not have considered this all along?

- **Understanding hardware is increasingly important to obtain efficient algorithms.**

Simple example

Second order

```
for i in arange(1,N-1):
  for j in arange(1,N-1):
    out[j+i*N] = inn[j+i*N] + alpha*(inn[j-1+i*N] - \
                   2.0*inn[j+i*N] + inn[j+1+i*N])
```

Fourth order

```
for i in arange(1,N-1):
  for j in arange(1,N-1):
    out[j+i*N] = inn[j+i*N] + alpha*( \
      -1./12.*inn[j-2+i*N] + 4./3.*inn[j-1+i*N] \
      -5./2.*inn[j+i*N] + 4./3.*inn[j+1+i*N] \
            -1./12.*inn[j+2+i*N]);
```

Code

```
python heat.py
Walltime (2ndord):  15.598
Walltime (4thord):  24.250
```

Simple example in C++

Second order

```
for(int i=1;i<N-1;i++)
    for(int j=1;j<N-1;j++)
        out[j+i*N] = in[j+i*N] + alpha*(in[j-1+i*N]
            - 2.0*in[j+i*N] + in[j+1+i*N]);
```

Timing

```
g++ -O3 heat.cpp
./a.out
Walltime (2ndord): 0.0067
Walltime (4thord): 0.0066
```

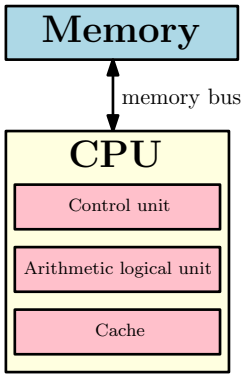
Both methods give the same performance.

A crash course in computer hardware

Components of a computer

CPU (central processing unit) performs arithmetic operations, conditionals, loops.

Memory stores data used for processing (main memory, caches, disk).



CPU and main memory is connected via the memory bus.

CPU

The CPU executes a sequence of instructions (referred to as machine code).

Example of a vastly simplified assembly/machine code

```
mov 0x38AF2 r1
mov 0xA03DD r2
add r1 r2
mov r2 0x38AF2
```

Instructions can be grouped as follows

- memory instructions (write or read from main memory)
- arithmetic operations on registers
- control instructions (comparisons, jumps)

CPU performance

CPU operates using a **clock rate**.

- One elementary operation is executed in each clock cycle.

Arithmetic performance is usually measured in floating point operations per second (**FLOPS**)

- amount of arithmetic operations that can (theoretically) be performed per second

A 3 GHz CPU that can perform one floating point operation per cycle has a performance of **3 GFLOPS**.

Moore's law

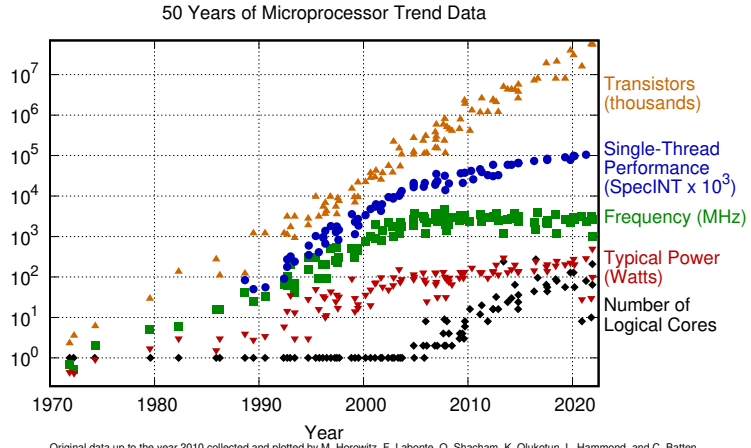
Moore's law

> *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. Over the longer term, the rate of increase is a bit more uncertain, although there is no reason to believe it will remain nearly constant for at least 10 years. – G. Moore, 1965*

or **the number of transistors per unit area doubles every 2 years**.

In popular culture a different version has prevailed: **Single threaded performance doubles every four years.**

CPU development



Plot from Karl Rupp (CC).

Modern CPUs

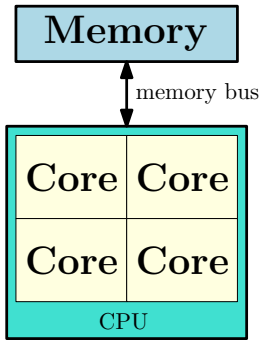
Increasing clock frequency has not been viable for a while

- power dissipation scales as the frequency squared
- but transistors still get smaller

Modern CPUs look more like this.

Multiple CPUs on the same chip

- usually the sequential execution units are referred to as cores
- CPUs with 16 cores are quite common now



Vectorization

Each core can perform vector operations in a single clock cycle

- 256 bit registers (4 double, 8 floats)
- fused multiply add

Vectorization is only possible if the CPU supports that specific instruction

- ideally handled by the compiler
- OpenMP includes support for vectorization

Types of parallelism

- Vectorization is single instruction multiple data (SIMD)
- Core level parallelism is multiple instruction multiple data (MIMD)

Performance of a modern CPU

The theoretically achieved FLOPS are calculated as follows

$$(3\ \text{GHz}) \cdot (16\ \text{cores}) \cdot (4\ \text{SIMD}) \cdot (2\ \text{fused multiply add}) \cdot (2\ \text{ALUs})$$
$$= 768\ \text{GFLOPS}$$

Factor of 256 **in performance if parallelization is fully utilized**.

Memory performance

Memory bandwidth is usually measured in Bytes transferred per second (GB/s).

- Amount of data that can (theoretically) be transferred to or from memory per second.

Typical value on modern computer systems is 100 GB/s.

Latency is also important (measured in seconds)

- Time it takes between a memory request is started until the data can be used by the CPU.

Typical value on modern computer systems is 50 ns.

- corresponds to a transfer of 5kB.

Compute bound vs memory bound

Memory bandwidth: 100 GB/s means we can **transfer 12G double precision** floating point numbers per second.

Compare to 768G additions or multiplications per second.

Let us consider multiplying a vector by a scalar

```
for(int i=0;i<n;i++)
    y[i] = 3*x[i]
```

Requires one memory read and one memory write per floating point operation.

- flop/byte ratio is 0.125 ($\ll$ than hardware flop/byte ratio of 7.68)

This is a **memory bound problem**.

Compute bound vs memory bound

A problem is **compute bound** if the performance is dictated by how many arithmetic operation the CPU can perform.

- numerical quadrature, solving dense linear system (LU), Monte Carlo methods, ...

A problem is **memory bound** if the performance is dictated by the bandwidth of main memory.

- stencil codes, solving sparse linear systems, FFT, ...
- performance measured in achieved GB/s

How many memory instructions?

```
for(int i=0;i<n-1;i++)
    y[i] = x[i+1] - x[i];
```

Memory hierarchy

FLOPS have increased dramatically but memory speed has lagged behind.

- there is a cost, capacity, speed trade-off involved

The result is a **memory hierarchy**

- Caches on the CPU (fast, tens of megabytes)
- Main memory (medium speed, tens of gigabytes)
- Disk storage (slow, terabytes)

Caches are usually further divided

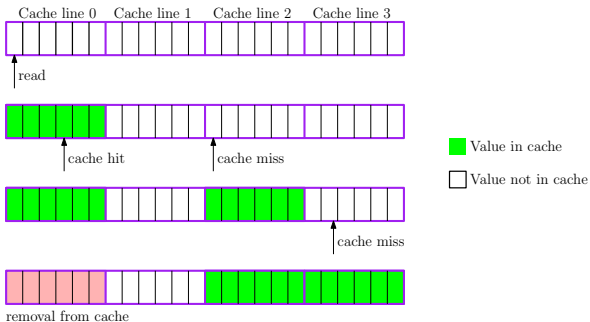
- Modern CPUs usually have L1, L2, L3 cache
- Caches are completely transparent to the programmer

Caches

Knowledge of how caches work is important for performance.

Caches transfer data in chunks of fixed size (so-called cache lines)

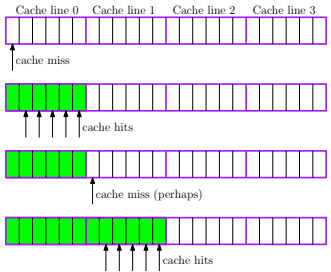
- usually 64-256 bytes in size (8-32 doubles)



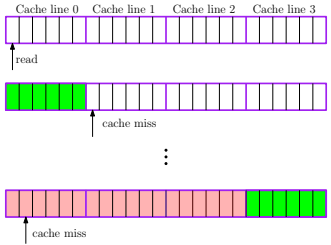
First read of any byte in a cache line transfers the entire cache line.

Memory access pattern

```
// Access with stride 1
for(int i=0;i<n;i++)
    out[i] = in[i];
```




```
// Access with stride 6
for(int j=0;j<6;j++)
    for(int i=0;i<n/6;i++)
        out[j+6*i] = in[j+6*i];
```



Latency

Latency – how long the CPU has to wait between issuing a memory request and receiving the first byte – is an important performance consideration.

- There are physical limitations when reducing latency

Modern CPUs employ a range of techniques to hide latency.

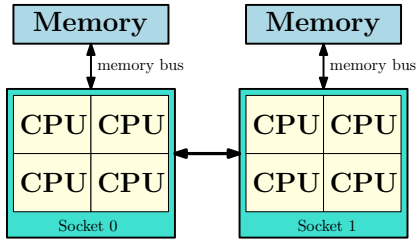
Prefetching tries to load data that is likely used next into the cache (before the actual memory instruction is issued)

- Particularly efficient for problems where memory locations close together are accessed in sequence

Instruction Level Parallelism (ILP) tries to skip instructions that still wait for memory.

NUMA domains

But many systems are dual or quad socket now.



All cores can access the entire memory but speed might differ depending on which memory modules are accessed.

- Non uniform memory access (NUMA)
- Cores are grouped into NUMA domains

Determine NUMA domains: `numactl --hardware`

First touch

For best performance memory has to be placed 'close' to where it is used.

- Neither C++ nor OpenMP provides a way to directly do that
- This might not necessarily be desirable anyhow

First touch principle: A memory location is mapped close to the core that first touches (reads or writes) it.

Different parts of an array can be placed on different NUMA domains.

A note on optimization

Modern CPUs are complicated.

- A basic understanding is vital but often measurement is necessary.

As a rule of thumb we pay the following penalty (in clock cycles)

Operation	cost in cycles
arithmetics	1
L1 hit	1-10
function call	10-20
L3 hit	40
sin/cos	100
memory	200
disk	10^5

Exact numbers depend on the specific architecture.

Back to the simple example

Second order (17 arithmetic operations, 2 memory operations)

```
out[j+i*N] = inn[j+i*N] + alpha*(inn[j-1+i*N] - \
                2.0*inn[j+i*N] + inn[j+1+i*N])
```

Flop/byte ratio is 1.1.

Fourth order (28 arithmetic operations, 2 memory operations)

```
out[j+i*N] = inn[j+i*N] + alpha*( \
  -1./12.*inn[j-2+i*N] + 4./3.*inn[j-1+i*N] \
  -5./2.*inn[j+i*N] + 4./3.*inn[j+1+i*N] \
        -1./12.*inn[j+2+i*N]);
```

Flop/byte ratio is 1.8.

Performance is only dictated by memory bandwidth.

- Both ratios are significantly below the hardware flop/byte ratio.
- Higher order space discretization is practically for free.

Supercomputing

Use case

Five-dimensional phase space (best case)

- $n = 100$: 160 GB of memory
- $n = 400$: 163 TB of memory

Six-dimensional phase space (best case)

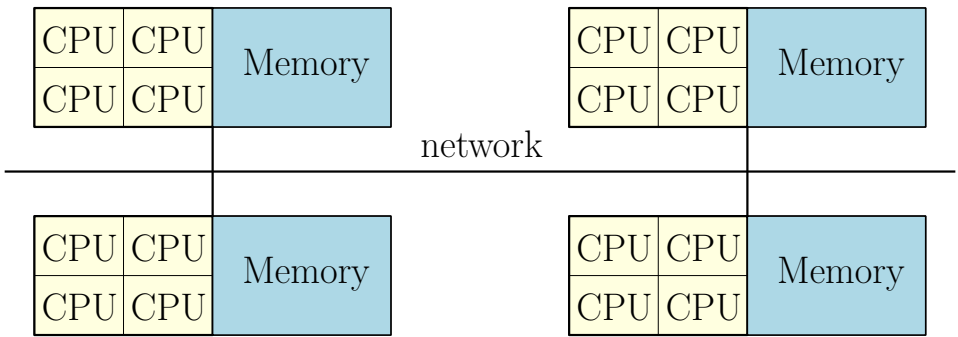
- $n = 50$: 250 GB of memory
- $n = 200$: 1024 TB of memory

Requires significantly more memory/computational power than a single computer can provide.

Cluster

Supercomputers are build by connecting many CPUs over a network.

- Modern supercomputers are build by connecting (via a network) a large number of 'commodity computers'.

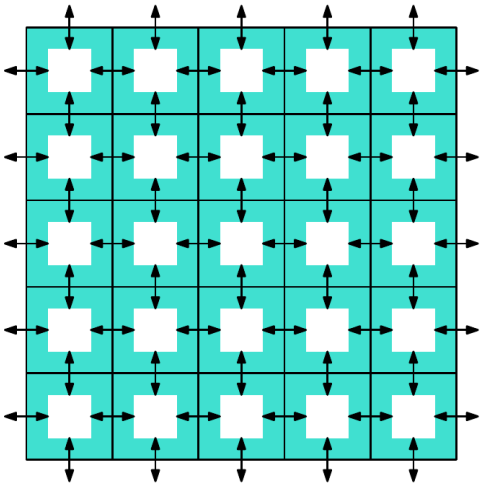


Communication over a network using the message passing interface (MPI).

Data distribution

Most scientific computing takes place on a domain $\Omega \subset \mathbb{R}^n$.

- We often have local data dependency (e.g. differential operators)
- Domain is divided into p (number of processes) usually equally sized parts



Data distribution

The **time loop** of a typical implementation

- send/receive boundary data from neighboring processes (at inter-process boundaries)
- perform the computation (independent on each process)
- repeat

Performance

Performance is dictated to a large extent by **how much data is communicated**.

FFT: to compute a Fourier coefficient

$$\hat{u}_k = \Delta x \sum_i u(x_i) \exp(\mathrm{i} k x_i)$$

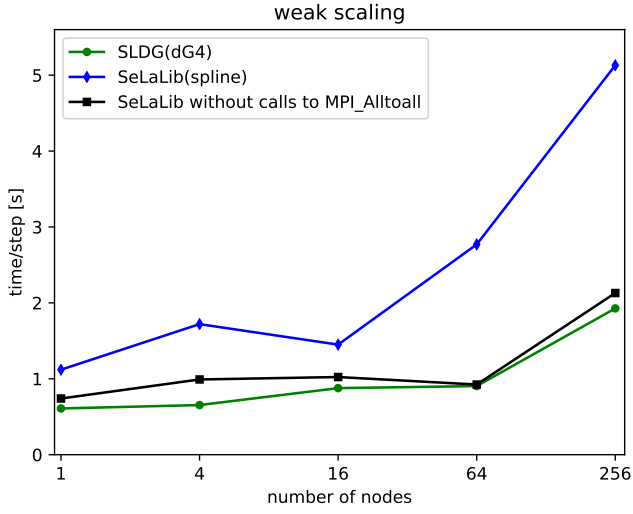
requires the knowledge of u at every grid point x_i – **global data dependency**.

Spline interpolation: Solving the linear systems to obtain the spline introduces a **global data dependency**.

Global data dependency requires that each process sends data to each other process (**All-to-all communication**).

Global communication

Global communication destroys performance.



Local communication

Supercomputers **favor algorithms that have local communication patterns.**

- The larger the number of processors the more advantage results from using local algorithms.

For semi-Lagrangian simulation the most well known local algorithms are

- Lagrange interpolation
- **Semi-Lagrangian discontinuous Galerkin** approach

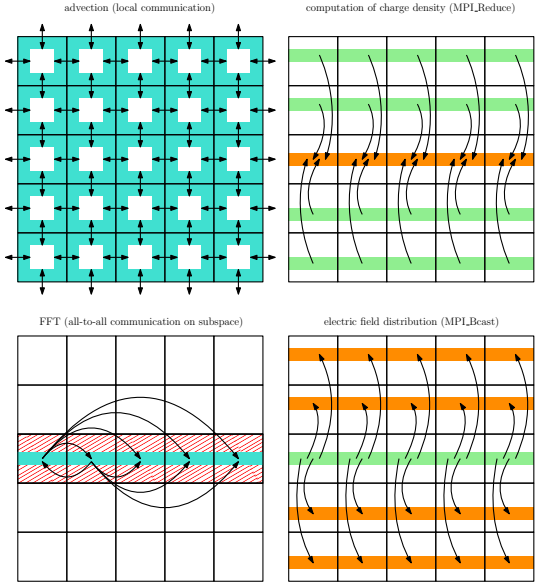
Work has been done on *localizing* spline interpolation.

N. Crouseilles, G. Latu, E. Sonnendrücker. J. Comput. Phys. 228:5, 2009.

Communication

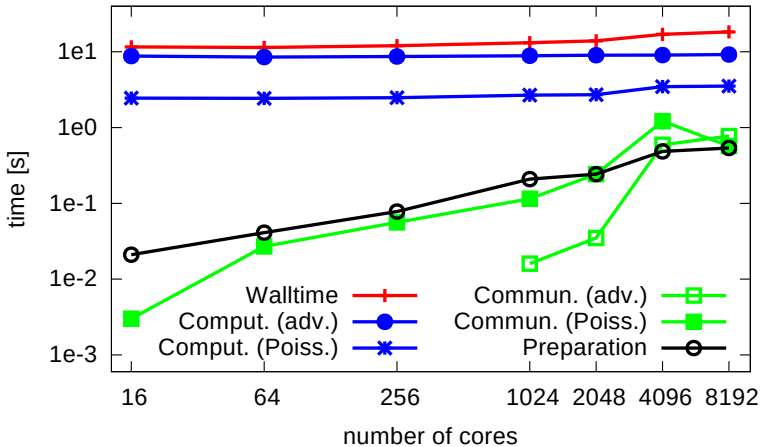
Typical communication pattern for a Vlasov–Poisson solver.

- Semi-Lagrangian advection: local communication
- Computing the charge density: reduction
- Poisson solver: all-to-all for a subset of processors.
- Electric field distribution: broadcast.



Scaling

Weak scaling for a discontinuous Galerkin based Vlasov–Poisson solver.



Poisson solver due to communication can become significant on very large systems.

Massively parallel architectures

Why massively parallel architectures

CPUs are built to accommodate a wide range of applications and environments. E.g. programs

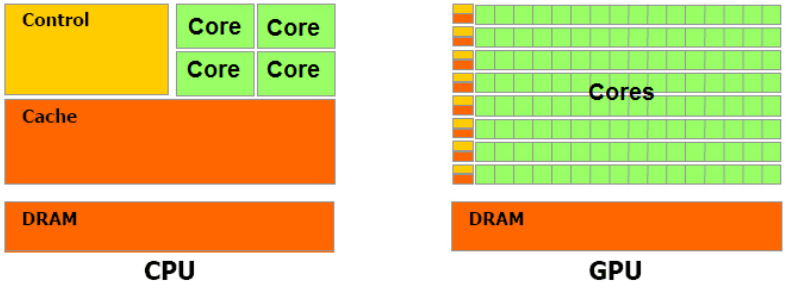
- that are not yet or can not be parallelized
- with very irregular memory access
- with highly variable workload

This **general purpose** architecture has a **performance cost** for scientific codes that are usually

- very well parallelizable
- have highly regular memory access
- perform the same operations over and over again

CPUs vs GPUs

GPUs (graphic processing units) are **massively parallel** chips ($>$ 2000 CUDA cores)



Best suited for computations with **high arithmetic intensity** or **predictable memory access patterns**

- many scientific codes fall into this category

Picture from NVIDIA (CC).

CPUs vs GPUs

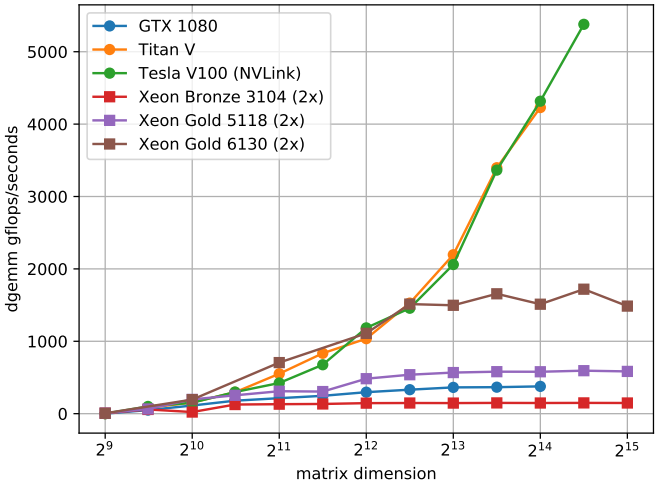
GPUs are assumed to be more difficult to program.

- But remember getting good performance from modern CPUs is also difficult

Parallelization is important even on CPUs.

- Even memory bound problems on CPUs require some parallelization.

CPUs vs GPUs (DGEMM)



Software aspects

Semi-Lagrangian dG scheme on GPUs

On GPUs the **choice of the algorithm is crucial**.

Semi-Lagrangian discontinuous Galerkin methods have the advantage of **completely local data dependency**.

Semi-Lagrangian discontinuous Galerkin is **not** just a **stencil code**

- non-aligned and (somewhat) unpredictable memory access;
- non-uniform degrees of freedom.

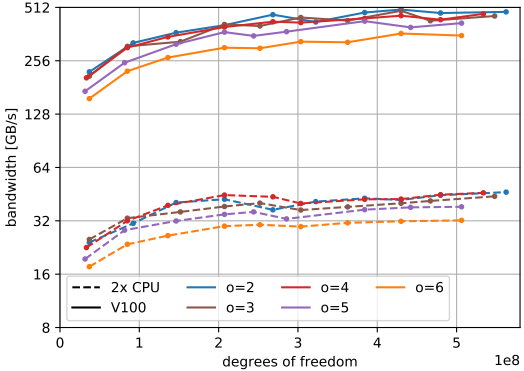
Performance results:

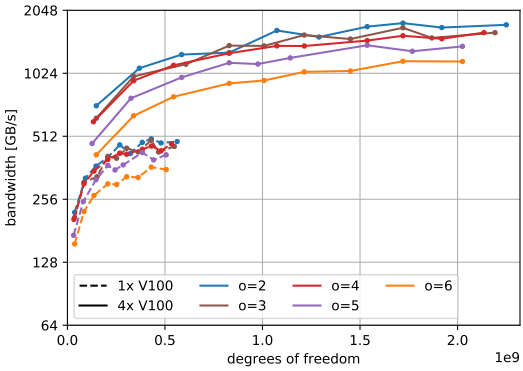
GPUs	1	4
TitanV	412 GB/s	0.98 TB/s
V100	**453 GB/s**	**1.60 TB/s**
A100	**729 GB/s**	**2.44 TB/s**

Implementation achieves 61%/73%/68% of peak performance.

Single GPU node outperforms entire LEO4 cluster! (96 CPUs)

Semi-Lagrangian dG scheme on GPUs





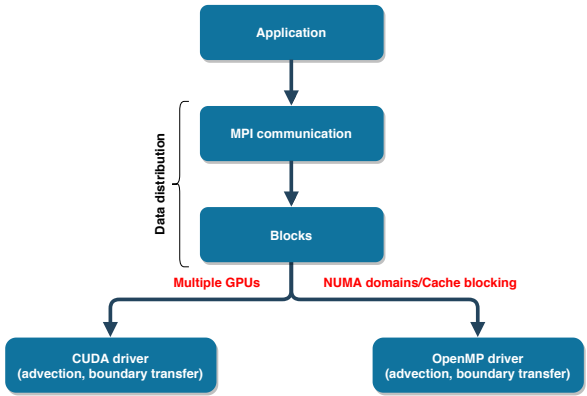
L.E. Comput. Phys. Commun. 254:107351, 2020.

SLDG software package

SLDG software package (MIT license)

- `https://bitbucket.org/leinkemmer/sldg`

Architecture



All levels are **independent**.

Some details

No reason why you can not **call a CUDA kernel on the CPU**.

```
#pragma omp parallel for schedule(static)
for(int i_omp=0;i_omp<gdim[2];i_omp++) {
    for(int iy=0;iy<gdim[1];iy++) {
        ...
        translate<dx,dv,dim,o>(...);
```

Writing **good GPU code** is also **beneficial on the CPU**.

For something more general: **cupla library**.

- Used e.g. in the PIConGPU code.

Good reasons for **keeping the driver code separate**

- Cache blocking on the CPU
- Boundary transfer on multiple GPU nodes

Some details

C++ templates are extensively used in **performance critical code**.

- Number of dimensions in x and v
- Order of the dG approximation

More important on the **CPU**.

Performance results for 4th order method (one node with 4x A100 on Juwels Booster)

dim	dof	without NVLink	with NVLink
2x2v	220^2440^2	1821 GB/s	**2437 GB/s**
2x3v	72^2144^272	1202 GB/s	**2567 GB/s**
3x3v	36^472^2	859 GB/s	**2052 GB/s**

GPU based supercomputing

The **fastest supercomputers in the world** are almost all **GPU based**.

- Performance and power efficiency of GPUs drives this development.

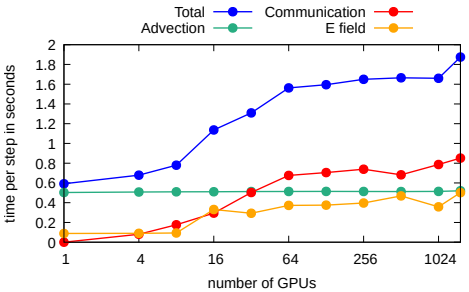
We are at **exascale** (10^{18} operations/s).

Frontier at Oak Ridge National Laboratory

- 37632 GPUs
- 1.1 EFlop/s and 9.2 PBytes of memory

Simulation with 1500 GPUs and $72^3 144^3$ grid points on JUWELS Booster:

- 2×24 AMD EPYC 7402 cores and $4\times$ **NVIDIA A100 GPUs per node**.
- Total of 150 TB of GPU memory.



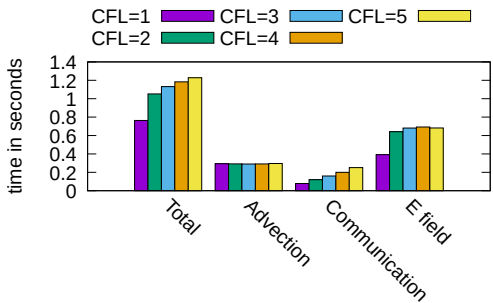
L.E., A. Moriggl. arXiv: 2110.14557.

Time step size

On **distributed memory** systems there is a **penalty for using larger time steps**.

- More data needs to be communicated in each step.

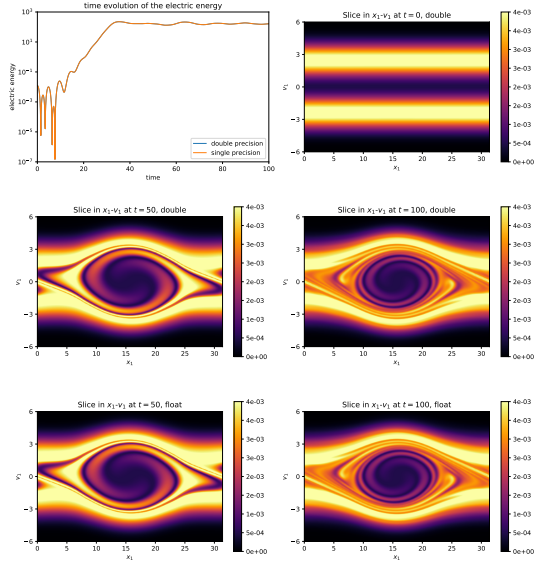
No more than 50% overhead in SLDG.



Single vs double precision

Single precision results in a reduction in memory and an increase in performance by **a factor of 2**.

- In many cases little difference in accuracy.
- Some care in the implementation is required.



Mixed precision computations

In each cell we store $c_0, \ldots, c_p$ where

$$u(x) = \sum_{k=0}^{p} c_k p_k(x).$$

For smooth solutions

$$c_0 = \mathcal{O}(1), \quad c_1 = \mathcal{O}(h), \quad \ldots \quad , \quad c_p = \mathcal{O}(h^p)$$

Store c_0 to c_m in double and c_{m+1} to c_p in single precision.

- Can be considered a lossy compression scheme.
- Exploits the particular structure of the dG approximation.
- How does half-precision fit in?

Mixed precision computations

Of particular importance

- c_0 is stored in double precision;
- $c_1, \ldots, c_p$ is stored in single precision.

Since

- charge is conserved up to double precision;
- memory usage is significantly reduced.

	order	# double	bandwidth	speedup	memorydown
NVIDIA K80	4	4	137.9 GB/s	–	–
	4	1	130.1 GB/s	1.51	1.60
	4	0	142.5 GB/s	2.07	2.00

Ensign

Ensign

Ensign is a framework for implementing dynamical low-rank algorithm on modern hardware systems.

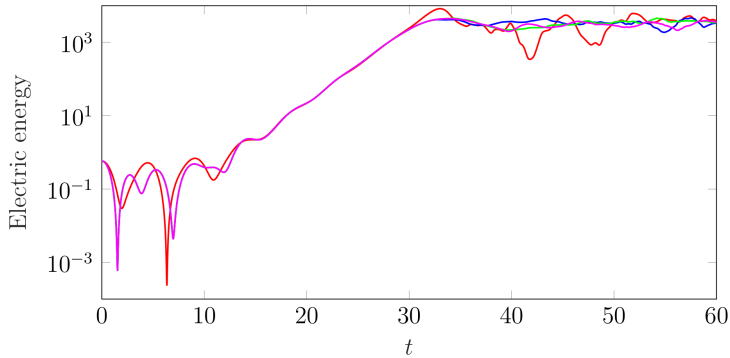
- Provides a collection of structures and functions to **easily implement dynamical low-rank approximation**.
- At the moment support for **multi-core CPUs** and CUDA capable **GPUs**.
- Uses highly optimized linear algebra routines behind the scenes (Intel MKL, cuBLAS, ...).
- Space and time discretization is left to the user.

Ensign is still under heavy development (MIT license)

- `https://github.com/leinkemmer/Ensign`

6D Vlasov simulation

A **3+3 dimensional two-stream instability** with $r = 5$, $r = 10$, $r = 15$, $r = 20$ using the **Ensign** framework (`https://github.com/leinkemmer/Ensign`).



Dense linear algebra is very efficient on GPUs.

- For $128^3 \times 128^3$ grid points and $r = 10$ simulation time on the GPU is 20 min.

Literature

Literature

[N. Crouseilles, G. Latu, E. Sonnendrücker. J. Comput. Phys. 228:5, 2009]

- Spline interpolation on patches that aids parallelization.

[J. Bigot et al. ESAIM: Proceedings 43, 2013]

- Scaling the semi-Lagrangian solvers Gysela on a Blue Gene system.

[K. Kormann, K. Reuter, M. Rampp. Int. J. High Perform. Comput. Appl. 2019]

- Lagrange based semi-Lagrangian Vlasov solver on large clusters.

[L.E. Comput. Phys. Commun. 254:107351, 2020]

- Details of the GPU implementation of the semi-Lagrangian discontinuous Galerkin scheme.

Literature

[L.E., A. Moriggl. arXiv:2110.14557]

- Scaling results on GPU based clusters for the semi-Lagrangian discontinuous Galerkin scheme.

[L.E. International Conference on High Performance Computing & Simulation (HPCS), 2016]

- Mixed precision Vlasov solver.

[F. Cassini, L. Einkemmer. arXiv:2110.13481]

- 6D Vlasov simulation with dynamical low-rank/Ensign.

SLDG: `https://bitbucket.org/leinkemmer/sldg`

Ensign: `https://github.com/leinkemmer/Ensign`