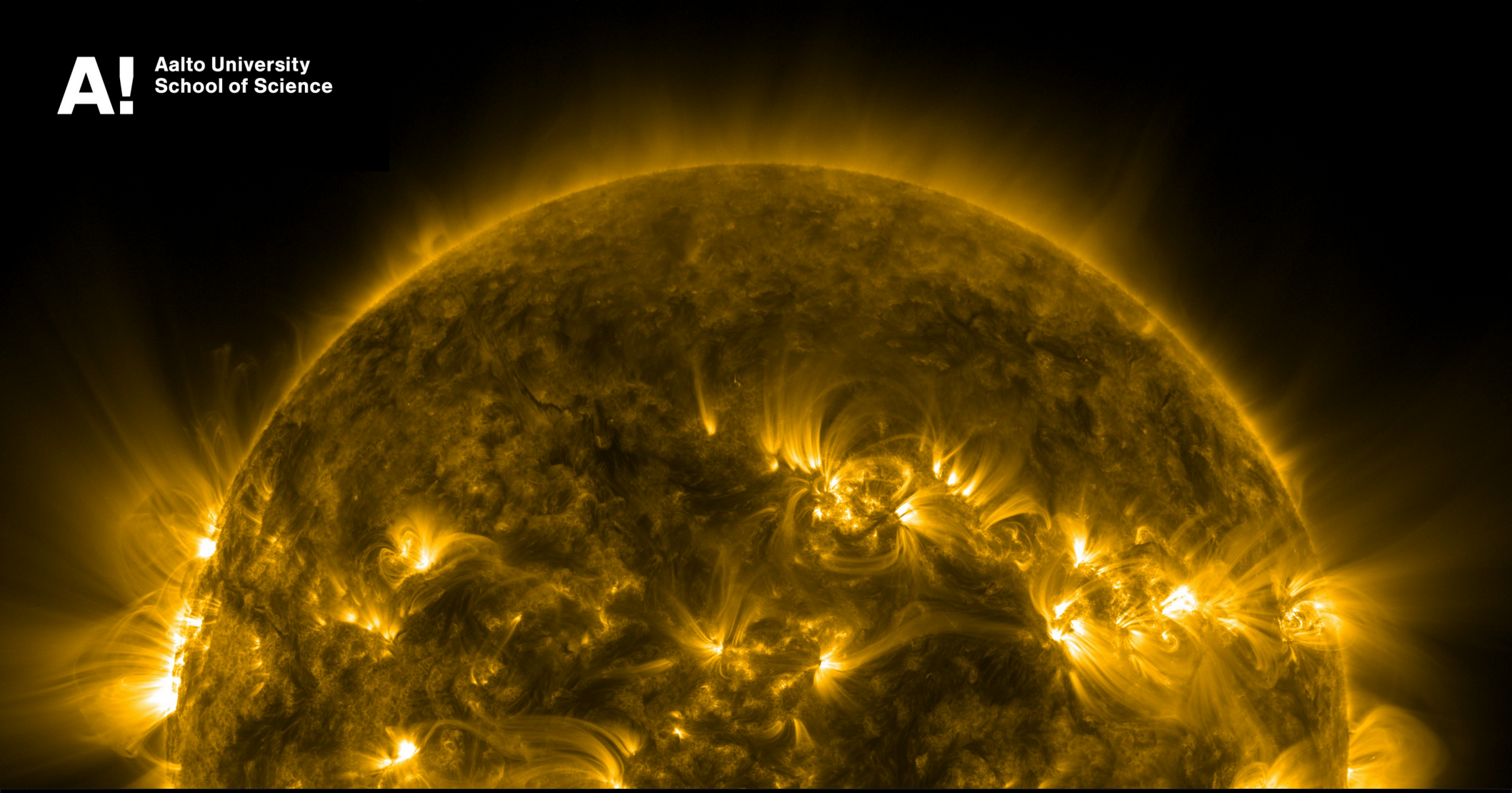
Aalto University
School of Science



Astaroth GPU PDE solver

AI Meets Plasma Physics, 22. Aug. 2022

Johannes Pekkila
Doctoral Researcher
Department of Computer Science, Aalto University

Image credit: NASA/SDO

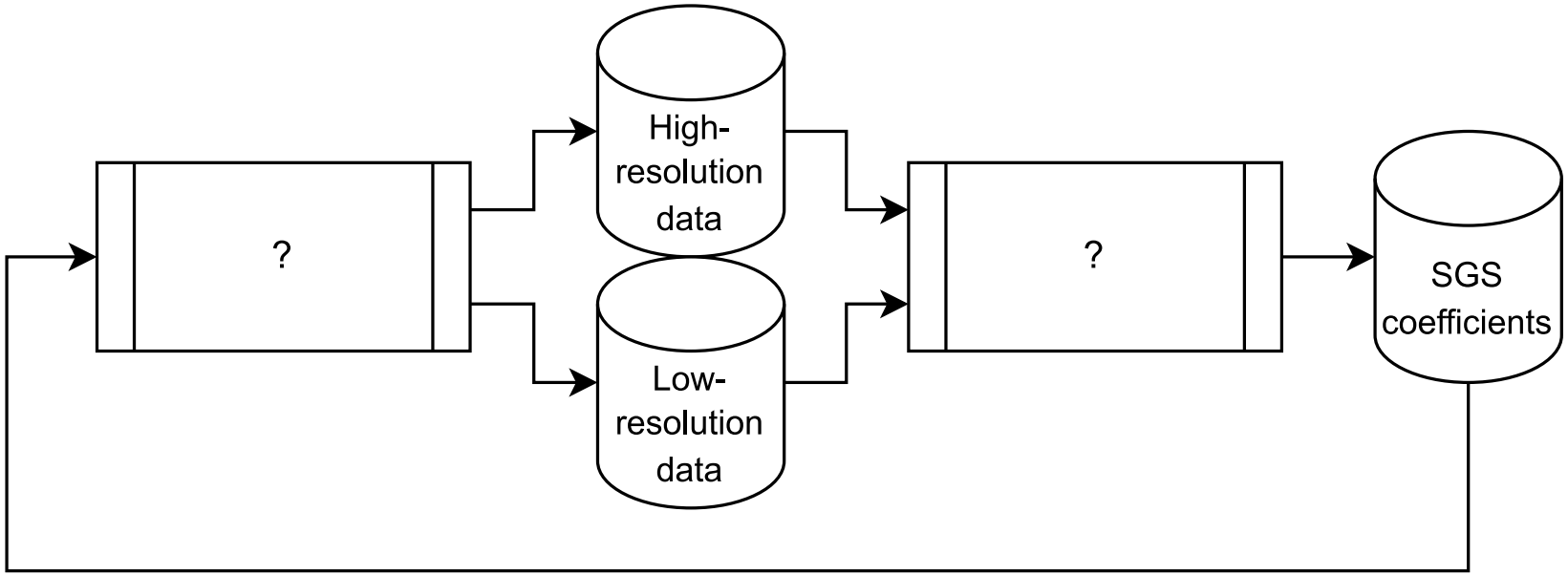
Agenda

- Introduction to Astaroth
- Astaroth DSL and API
- Guided tutorial: Blur filter
- Guided tutorial: Hydro
- Guided tutorial: Adding SGS stress
- Workshop: hands-on exercises

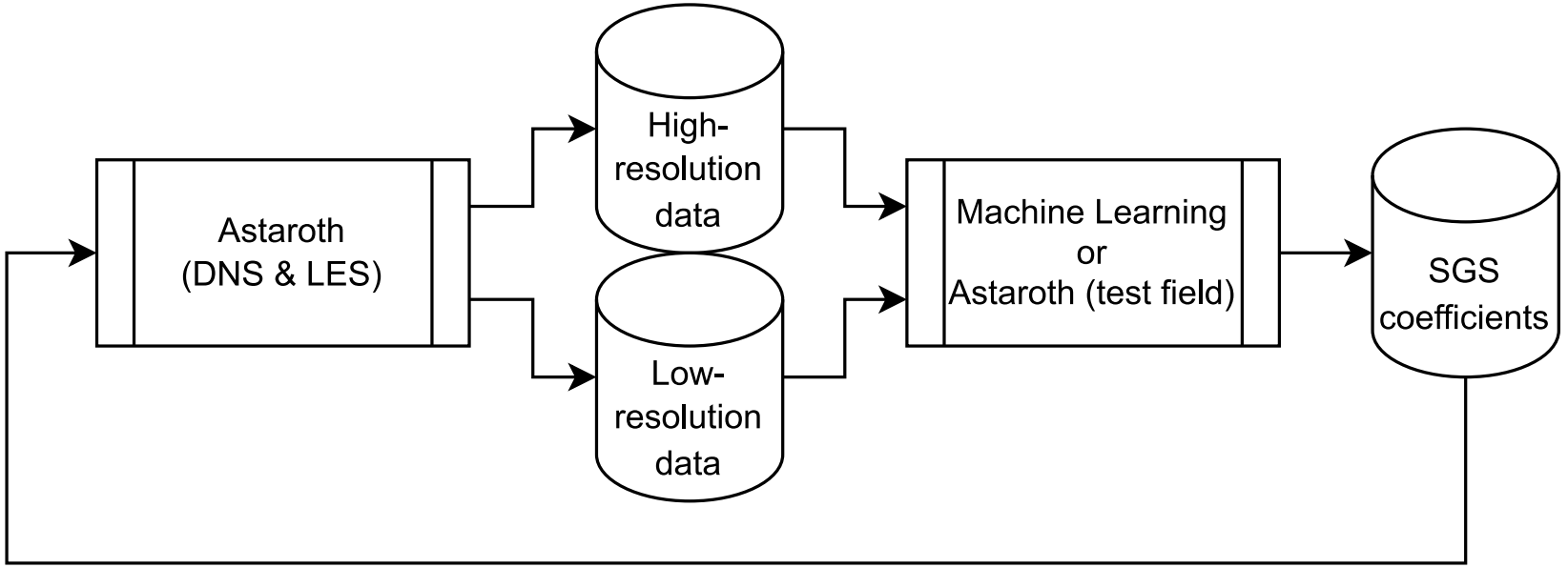
Introduction

- Astaroth - A GPU framework for stencil computations
  - Tuned especially for accelerating computational physics (high-order stencils, several coupled fields)
  - PDEs (e.g. finite differences), convolution kernels, others
  - Support for NVIDIA and AMD GPUs
  - Support for MPI, MPI IO, and GPUDirect RDMA
- C/C++ API
  - For controlling host-side execution, concurrency
- A domain-specific language for writing kernels
  - Syntax similar to Python
  - Translated into cache-efficient CUDA/HIP kernels at compilation

The Big Picture



The Big Picture



Introduction

- Why Astaroth?
  - Flexible
    - Designed to be used for any stencil-like computation
    - Easily adapted to LES + SGS with the DSL (this tutorial)
  - Efficient
    - Performance within an order of magnitude of the hardware maximum (assuming ideal caching and a perfect algorithm)
    - Notably faster than traditional CPU computations

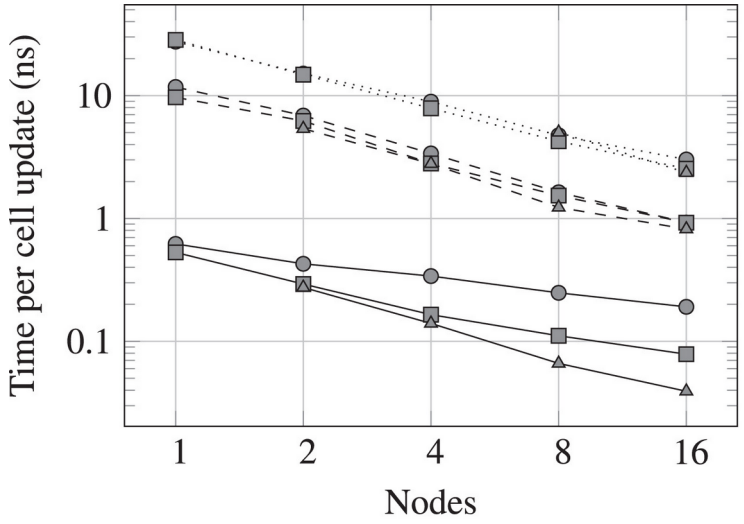
Performance

- Systems
  - 4x NVIDIA V100 GPUs (2017, Puhti)
  - 2x 64-core AMD Rome 7H12 64-core CPUs (2019, Mahti)

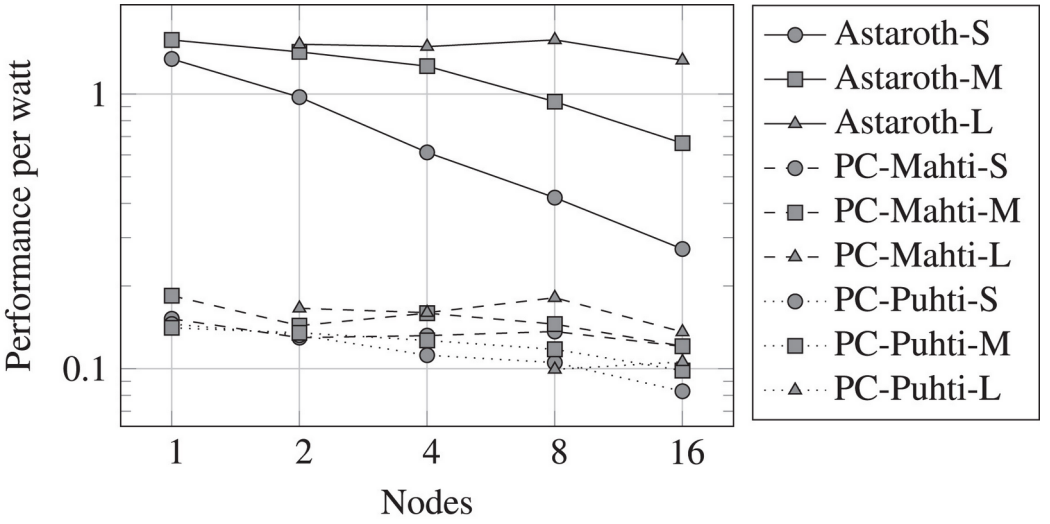
- Speedups compared to a multi-core CPU solver (MHD)
  - 18x improvement per node
  - 20x improvement per 16 nodes

- Energy efficiency (perf. per Watt)
  - 9x improvement

Performance



(a) Strong scaling.



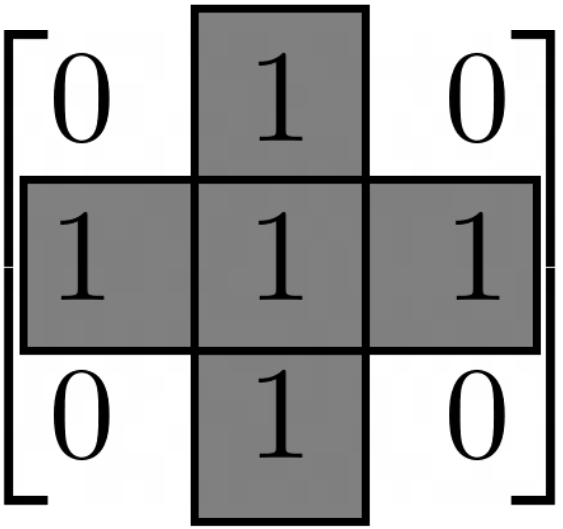
(b) Performance per watt (10^6 cell updates per second per watt).

Introduction

- Drawbacks of Astaroth
  - Small team
    - Limited manpower to fix bugs, introduce new features
  - Can still be difficult to get things working despite simplified programming
    - Most of the API is asynchronous by design due to performance considerations
    - Error messages originating from DSL-CUDA interaction can be cryptic at times

Stencil Computations

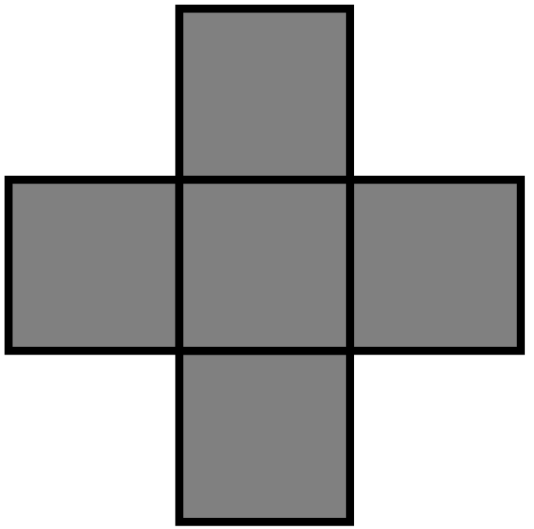
- Compute a single output from neighboring inputs
- Apply iteratively to all elements in a grid
- Stencil a visual representation of the convolution kernel



Stencil Computations

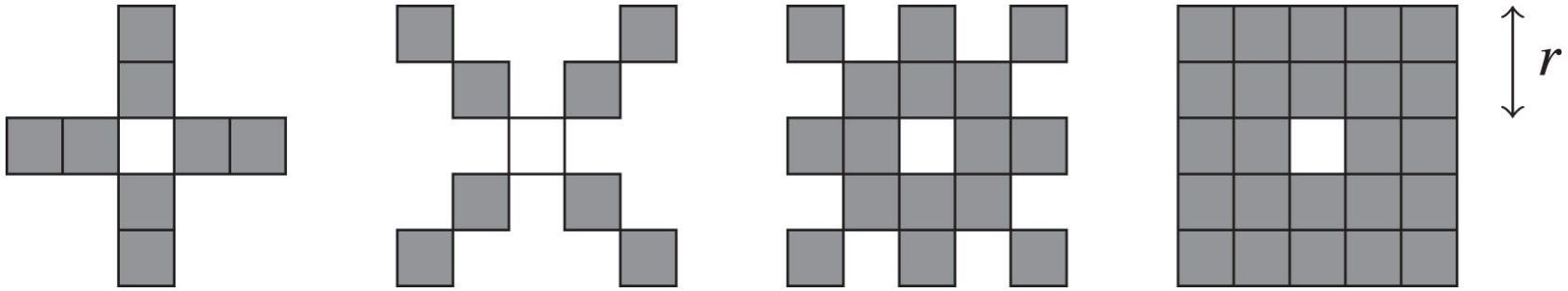
- Compute a single output from neighboring inputs
- Apply iteratively to all elements in a grid
- Stencil a visual representation of the convolution kernel

$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & 1 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

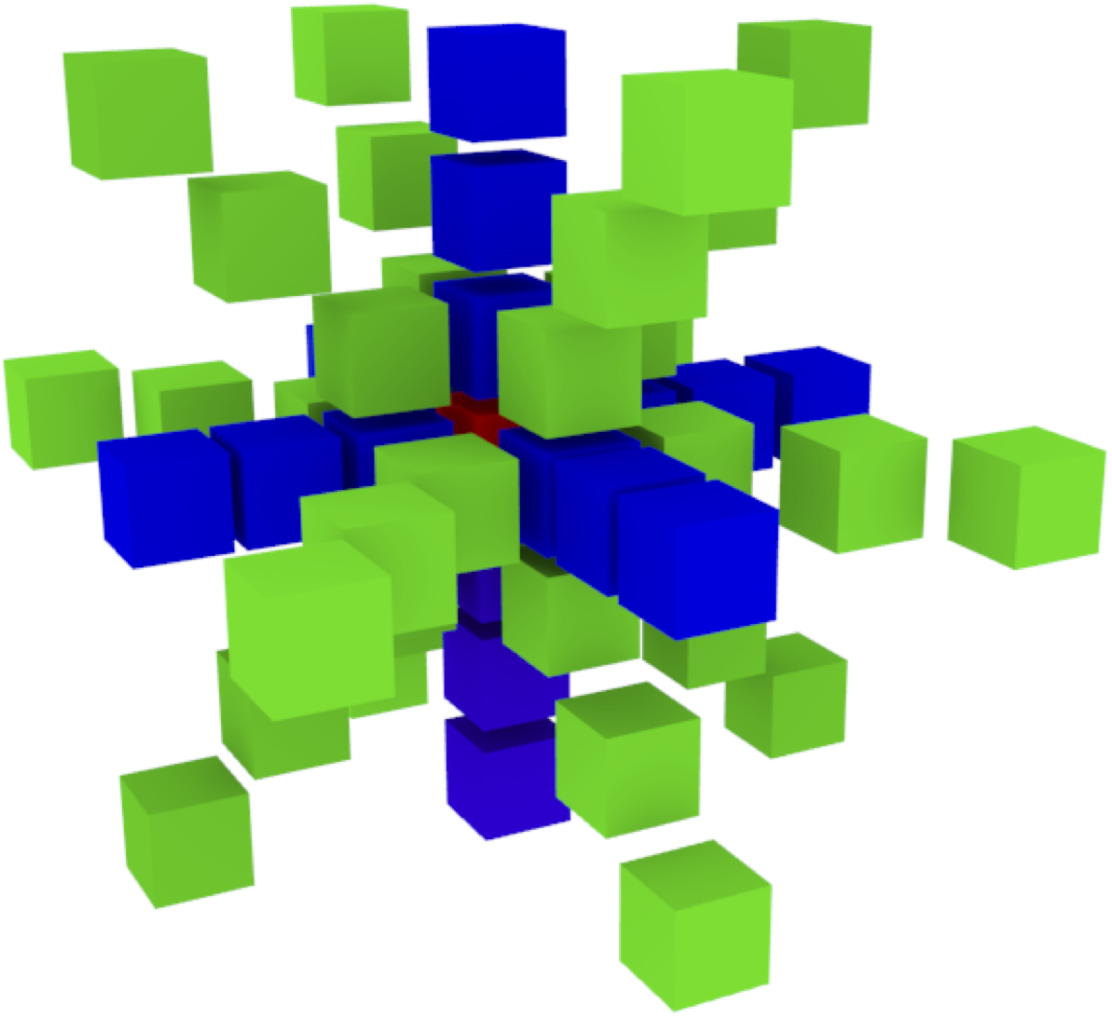


Stencil Computations

- Compute a single output from neighboring inputs
- Apply iteratively to all elements in a grid
- Stencil a visual representation of the convolution kernel



6^{th} order finite-difference stencil



Stencil Computations

- Status quo: The compute-to-bandwidth ratio of modern multiprocessors is large and increasing
  - V100: 36 floating-point operations per data word served
  - MI250X: 117 floating-point operations per data word
- Issue #1: how can we utilize all the available performance?
- Optimal algorithm: cache the working set

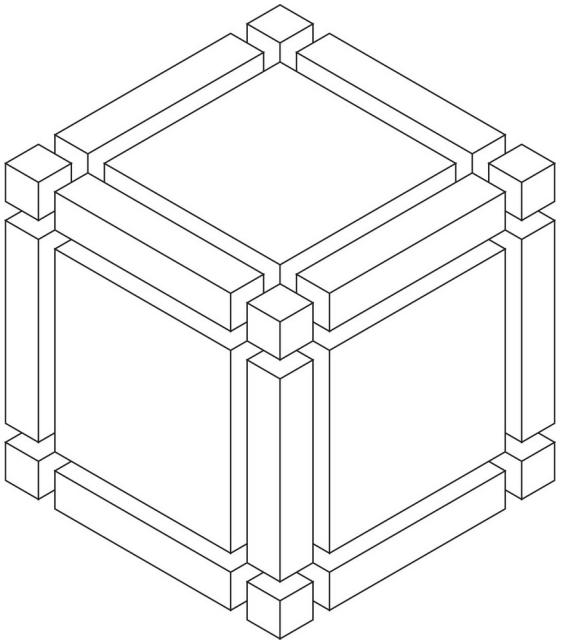
Stencil Computations

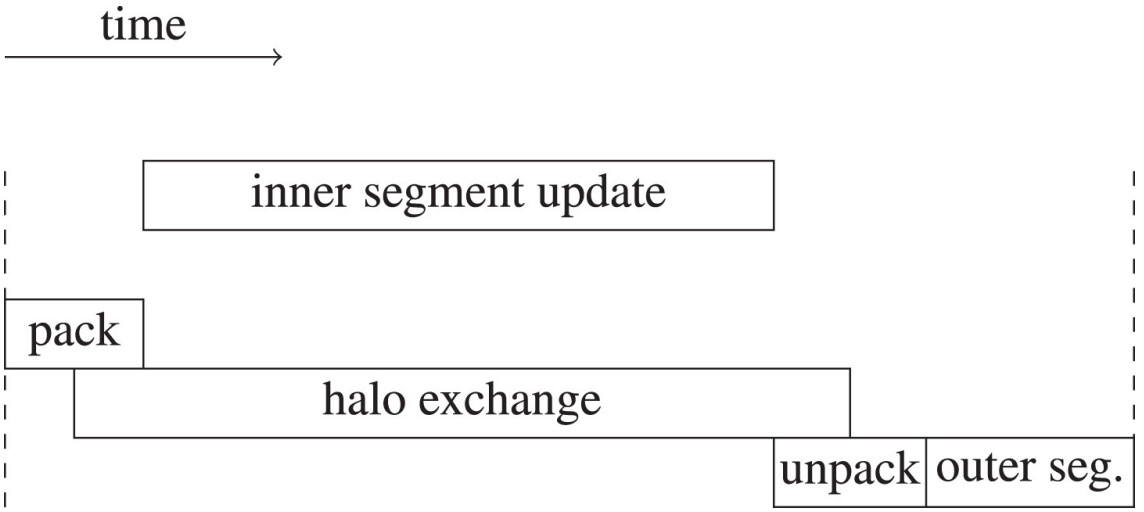
- Issue #2: How can we fit everything in caches?
  - Process fields one at a time to cache what you can and store intermediate results in local memory (cache oblivious)
  - Reorder instructions to maximize instruction-level parallelism (latency hiding)

- Requires manual unrolling and code reordering to work well
  - This is what our DSL compiler does
  - Full MHD
    - ~500 lines of code (DSL)
    - ~5000 lines of code (CUDA)

Stencil Computations

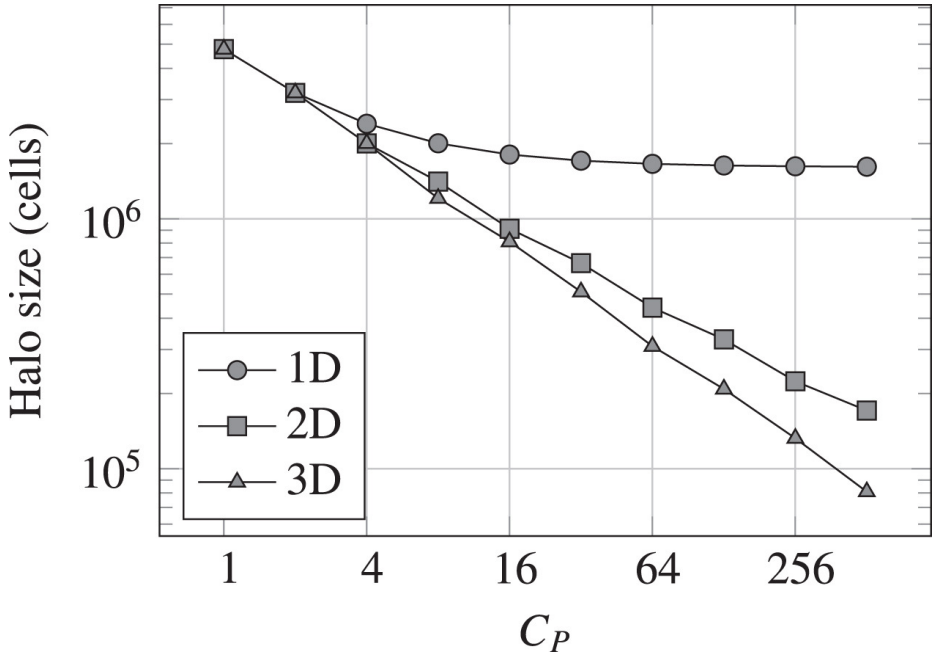
- Issue #3: How can we get it to scale?
  - Perform computation and communication in parallel





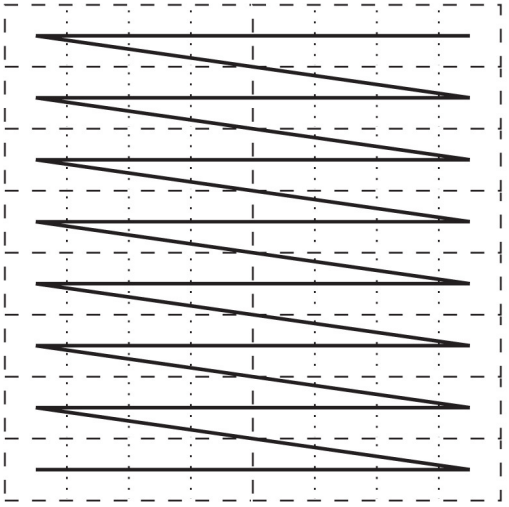
Stencil Computations

- Issue #3: How can we get it to scale?
  - Perform computation and communication in parallel
  - Minimize communication surface by 3D decomposition

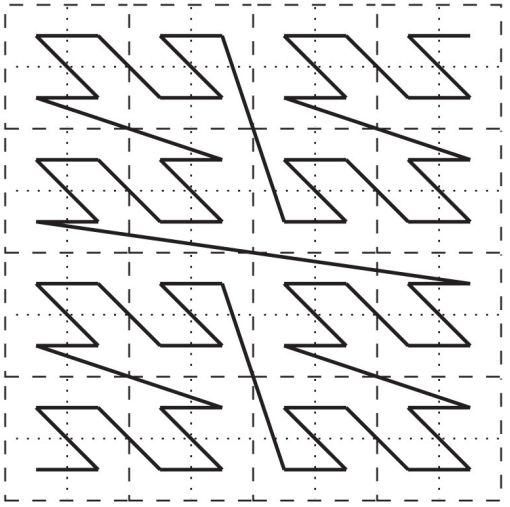


Stencil Computations

- Issue #3: How can we get it to scale?
  - Perform computation and communication in parallel
  - Minimize communication surface by 3D decomposition
  - Maximize data locality by Z-order processor assignment



(a) Row-wise scan.



(b) Z-order indexing.

EMPTY SLIDE

Outlook (Matthias)

- Astaroth can be utilized embedded in multiphysics simulation package Pencil Code as its core solver.

Embedding

- Provides the full spectrum of Pencil Code
  - Initial conditions
  - Boundary conditions (as long as not kernel-reimplemented) grids
  - On-the fly diagnostics: spectra, averages (0D to 2D), extrema, structure functions, performance indicators
  - Visualization data (e.g. slices, streamlines)
  - IO schemes (e.g., collective FORTRAN binary, MPI, HDF5)

Embedding, continued

- Provides the full surrounding infrastructure of Pencil Code for
  - Code building setup generation
  - Management data analysis
  - Visualization (Python, IDL, Mathematica packaes)
  - Auxiliary functions for restarting, regridding, data collection and testing, archiving
- Pencil Code embedding
  - Alleviates lack of manpower: developers can concentrate on performance
  - Improvement of the Astaroth core; Astaroth usage can immediatley profit from Pencil Code enhancements.
  - Lowers acceptance threshold for users focussed on production
  - Is now in test and optimisation stage.

Embedding, continued

- simplifies use of DSL by automatic generation of standardized code:
  In the simplest case, the user fills only the bodies of the rhs functions in
  an automatically generated template (equations.h), e.g. for standard MHD:

```
// Generated by Pencil Code build, yet meant to be edited by the user.
// All needed functions are provided, but unneeded ones are not removed.

dUU_dt(){ }
dSS_dt(){ }
dRHO_dt(){ }
dAA_dt(){ }
```

by hand like

```
dAA_dt{
  return cross(vecvalue(UU), curl(AA)) + AC_eta * veclaplace(AA)
}
```

or using predefined code-snippets like

```
DAA_DT{
# include "induction_hall.h"   // Hall EMF included
}
```

Host paramters like eta are automatically conveyed to GPU memory.

EMPTY SLIDE

Getting Started

- Astaroth is freely available under GPLv3
  https://bitbucket.org/jpekkila/astaroth
- See `astaroth/samples/plasma-meets-ai-workshop`

Getting started: DSL

- Workflow

  1) Define fields

```
1  Field lnrho
2  Field ux, uy, uz
3  #define uu Field3(ux, uy, uz)
```

Getting started: DSL

- Workflow
  1) Define fields
  2) Define stencils

$$f_i \approx \frac{1}{2h} f_{i+1} - \frac{1}{2h} f_{i-1}$$

```
#define h (...)
Stencil ddx {
    [0][0][1] = 1.0/(2 * h),
    [0][0][-1] = -1.0/(2 * h)
}
```

Getting Started: DSL Stencils

```
Stencil identifier {
    [k_0][j_0][i_0] = c_0,
    ...,
    [k_n][j_n][i_n] = c_n,
}
```

$$f'(x,y,z) = c_0 f(x+i_0, y+j_0, z+k_0) + ... + c_n f(x+i_n, y+j_n, z+k_n)$$

Getting started: DSL

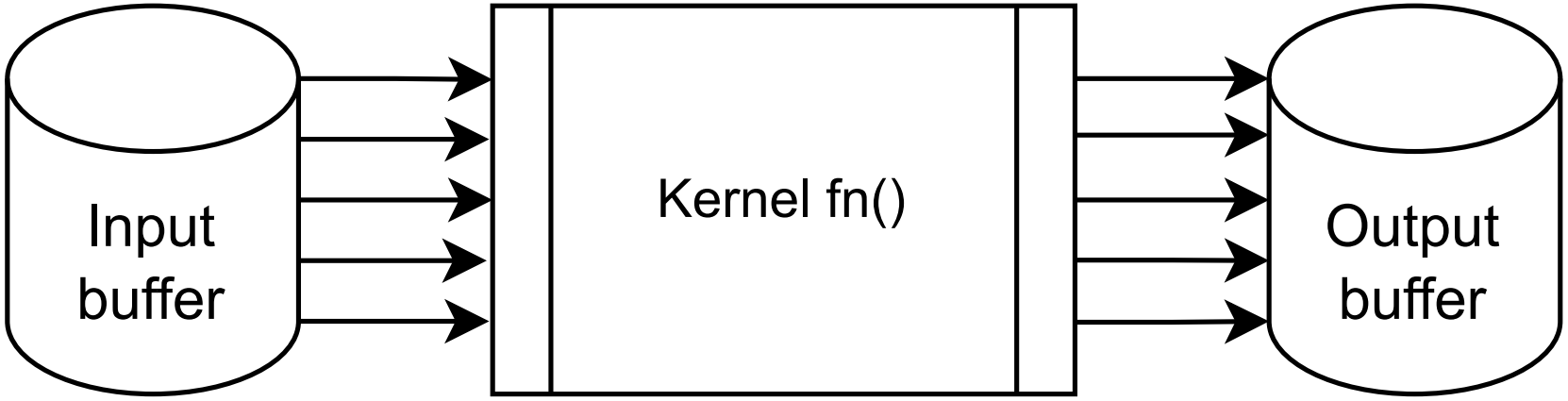
- Workflow
  1) Define fields
  2) Define stencils
  3) Define kernels

```
Kernel fn() {
    write(lnrho, ddx(lrnho))
}
```

Getting started: DSL

- DSL paradigm: Dataflow/stream programming

```
Kernel fn() {
    write(lnrho, ddx(lrnho))
}
```



Getting started: DSL program

```
Field lnrho
Field ux, uy, uz // Unused
#define uu Field3(ux, uy, uz)

#define h (1.0)
Stencil ddx {
    [0][0][1]  =  1.0 / (2*h),
    [0][0][-1] = -1.0 / (2*h)
}

Kernel fn() {
    write(lnrho, ddx(lrnho))
}
```

Getting Started: API

- Include Astaroth headers

```
#include "astaroth.h"        // The main library
#include "astaroth_utils.h" // Utility functions, such as `acLoadConfig`
```

Getting Started: API

- Include Astaroth headers
- Load the mesh configuration

```
AcMeshInfo info;
acLoadConfig(AC_DEFAULT_CONFIG, &info);
```

Getting Started: API

- Include Astaroth headers
- Load the mesh configuration
- Create a device

```
Device device;
acDeviceCreate(0, info, &device);
```

Getting Started: API

- Include Astaroth headers
- Load the mesh configuration
- Create a device
- Launch DSL kernels and synchronize as needed

```
for (size_t i = 0; i < num_iterations; ++i) {
    // Compute
    acDeviceLaunchKernel(device, STREAM_DEFAULT, fn, dims.n0, dims.n1);
    acDeviceSwapBuffers(device);
    acDevicePeriodicBoundconds(device, STREAM_DEFAULT, dims.m0, dims.m1);
}
acDeviceSynchronizeStream(device, STREAM_DEFAULT);
```

Getting Started: API

- Include Astaroth headers
- Load the mesh configuration
- Create a device
- Launch DSL kernels
- Deallocate memory at exit

```
acDeviceDestroy(device);
```

Empty slide

Getting started: IO

- Many ways to set initial conditions and write data to/from the GPU
- In production one would use MPI IO
  - Already implemented in Astaroth (acGrid functions)
- This tutorial: just quick & simple functions for transferring data between device-host-disk

Getting Started: Loading data

```
AcMesh mesh;
acHostMeshCreate(info, &mesh);
acHostMeshRandomize(&mesh);
acDeviceLoadMesh(device, STREAM_DEFAULT, mesh);
```

Getting Started: Storing data

```
for (size_t i = 0; i < num_iterations; ++i) {
    // Compute
    // ...

    // Store to host memory and write to a file
    acDeviceStoreMesh(device, STREAM_DEFAULT, &mesh);
    acDeviceSynchronizeStream(device, STREAM_DEFAULT);
    acHostMeshWriteToFile(mesh, i);
}
```

Getting Started – Full Program

```c
#include "astaroth.h"
#include "astaroth_utils.h"

int
main(void)
{
    // Setup the mesh configuration
    AcMeshInfo info;
    acLoadConfig("../samples/plasma-meets-ai-workshop/astaroth.conf", &info);

    // Allocate memory on the GPU
    Device device;
    acDeviceCreate(0, info, &device);
    acDevicePrintInfo(device);

    const AcMeshDims dims = acGetMeshDims(info);

    // Setup initial conditions
    AcMesh mesh;
    acHostMeshCreate(info, &mesh);
    acHostMeshRandomize(&mesh);
    acDeviceLoadMesh(device, STREAM_DEFAULT, mesh);

    // Write the initial snapshot to a file
    acHostMeshWriteToFile(mesh, 0);
```

```c
    // Ensure boundaries are up to date before starting computations
    acDevicePeriodicBoundconds(device, STREAM_DEFAULT, dims.m0, dims.m1);
    for (size_t i = 1; i < 20; ++i) {
        // Compute
        acDeviceLaunchKernel(device, STREAM_DEFAULT, fn, dims.n0, dims.n1);
        acDeviceSwapBuffers(device);
        acDevicePeriodicBoundconds(device, STREAM_DEFAULT, dims.m0, dims.m1);
        acDeviceSynchronizeStream(device, STREAM_DEFAULT);

        // Store to host memory and write to a file
        acDeviceStoreMesh(device, STREAM_DEFAULT, &mesh);
        acDeviceSynchronizeStream(device, STREAM_DEFAULT);
        acHostMeshWriteToFile(mesh, i);
    }

    // Deallocate memory on the GPU
    acDeviceDestroy(device);
    acHostMeshDestroy(&mesh);
    return EXIT_SUCCESS;
}
```

References

Code
http://bitbucket.org/jpekkila/astaroth

Publications

DSL (Pekkila 2019) http://urn.fi/URN:NBN:fi:aalto-201906233993

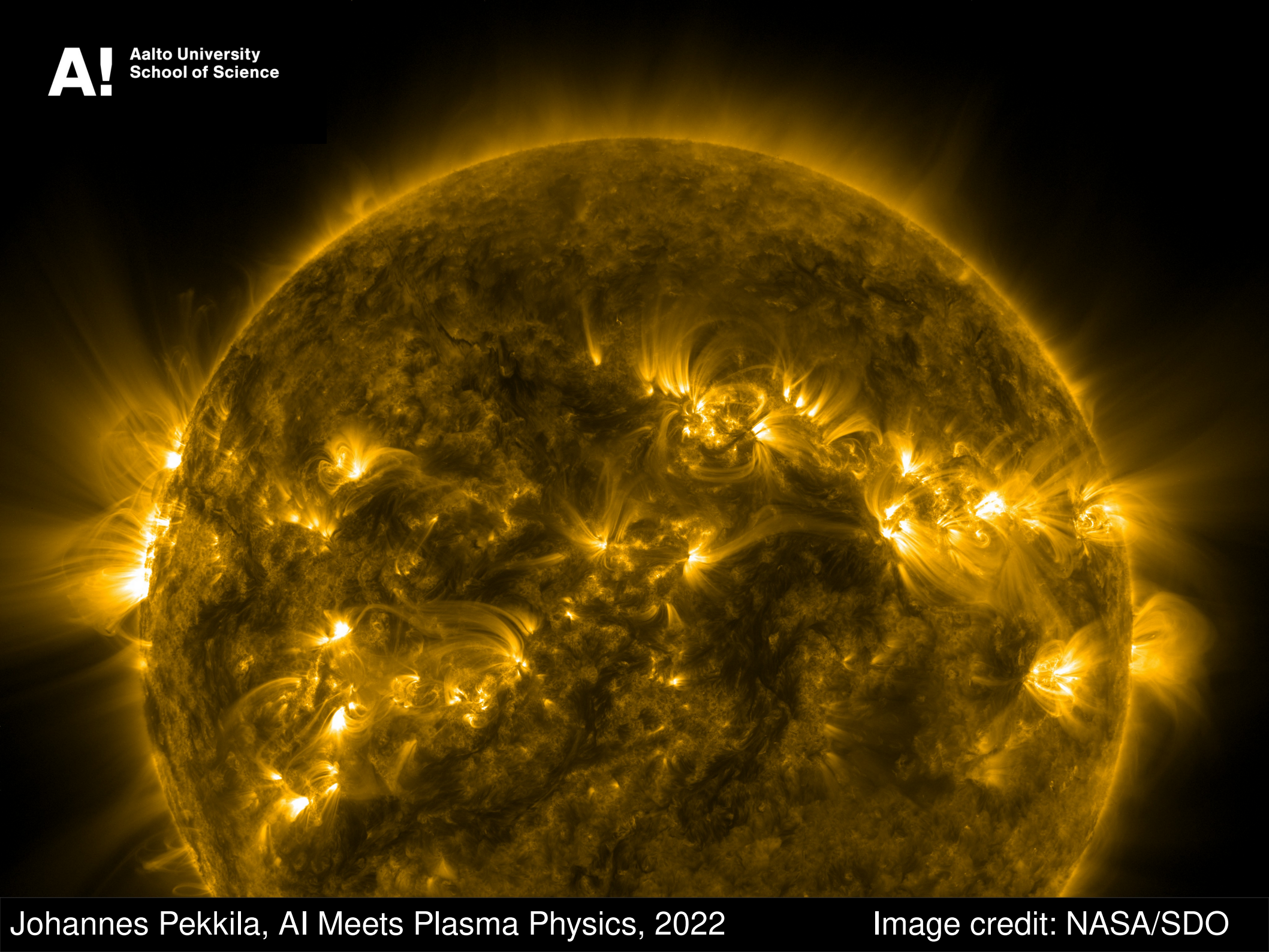
Physics (Vaisala 2020) https://arxiv.org/abs/2012.08758

MPI (Pekkila 2021) https://arxiv.org/abs/2103.01597

Task Scheduler (Lappi 2021) http://urn.fi/URN:NBN:fi-fe2021050428868

Special thanks

M. Väisälä
M. Käpylä
M. Rheinhardt
O. Lappi
Aalto University Astroinformatics Group
CSC – IT Center for Science
ERC Consolidator Grant UniSDyn


Aalto University
School of Science
Johannes Pekkila, AI Meets Plasma Physics, 2022
Image credit: NASA/SDO

Exercise 1

- Implement a blur filter with the DSL and visualize results

$$f'(x,y,z) = \frac{1}{9} \sum_{j=-1}^{1} \sum_{i=-1}^{1} f(x+i, y+j, z+k)$$

- See `astaroth/samples/plasma-meets-ai-workshop/README.md` for more details

Exercise 2

- Implement the continuity equation for compressible hydro simulation

$$D/Dt = \partial/\partial t + (\mathbf{u} \cdot \nabla)$$

$$\frac{D \ln \rho}{Dt} = -\nabla \cdot \mathbf{u}$$

- See `astaroth/samples/plasma-meets-ai-workshop/README.md` for more details

Exercise 3

- Implement the Smagorinsky SGS stress term and compute its divergence

$$\tau_{ij}^{smag} = -2(c_s \Delta)^2 |\overline{\overline{S}}| \overline{\overline{S}}_{ij}$$

$$-\nabla \cdot \tau$$

- See `astaroth/samples/plasma-meets-ai-workshop/README.md` for more details

EMPTY SLIDE

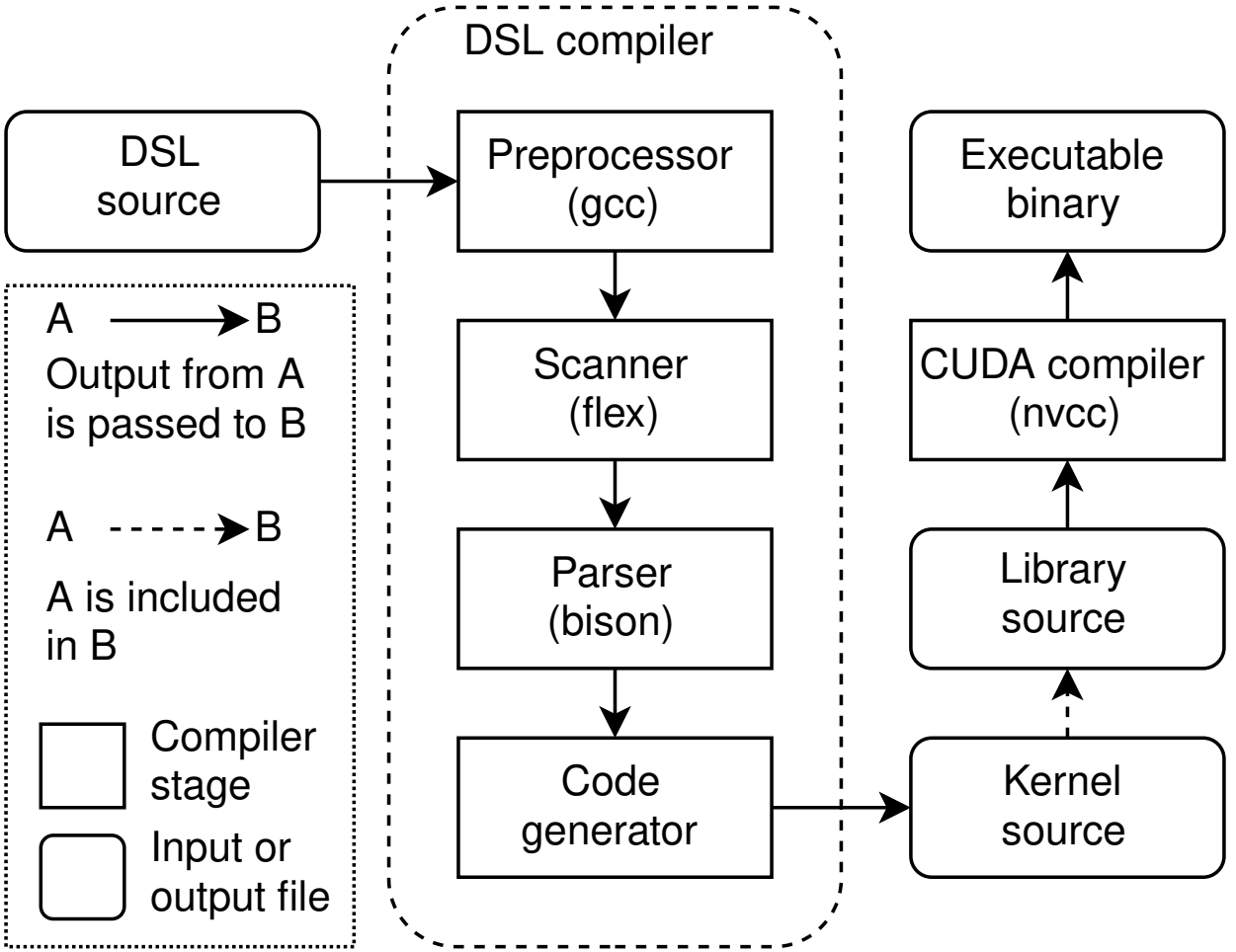
DSL overview

- https://bitbucket.org/jpekkila/astaroth/src/master/acc-runtime/README.md
- Fields, Stencils
- Data-flow programming: executed on all grid points

API overview

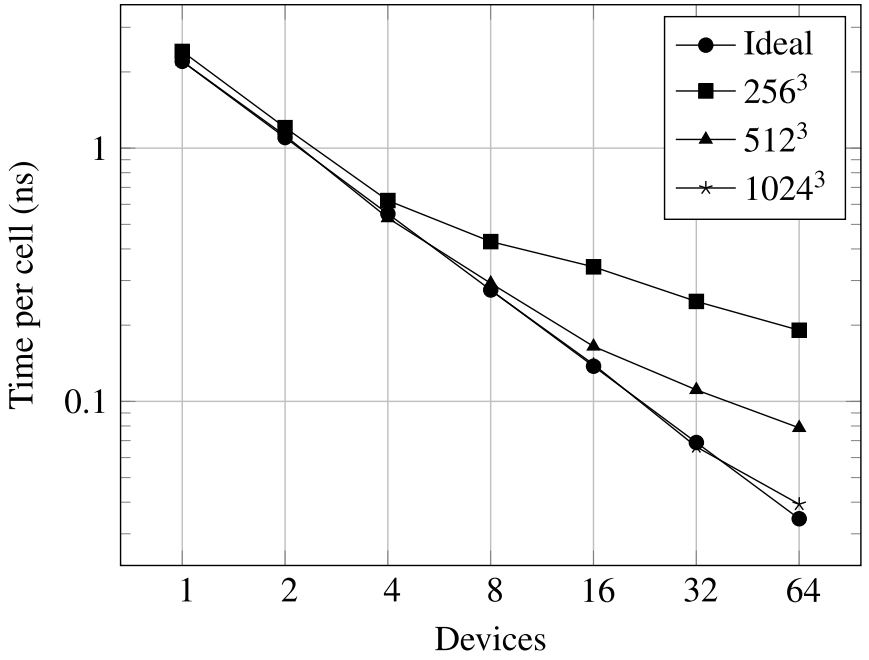
- https://bitbucket.org/jpekkila/astaroth/src/master/doc/Astaroth_API_specification_and_user_manual/API_specification_and_user_manual.md
- AcDeviceCreate, acDeviceDestroy, acDeviceLoad, acDeviceStore, acDeviceLaunchKernel, acDeviceSynchronizeStream

Stencil Computations

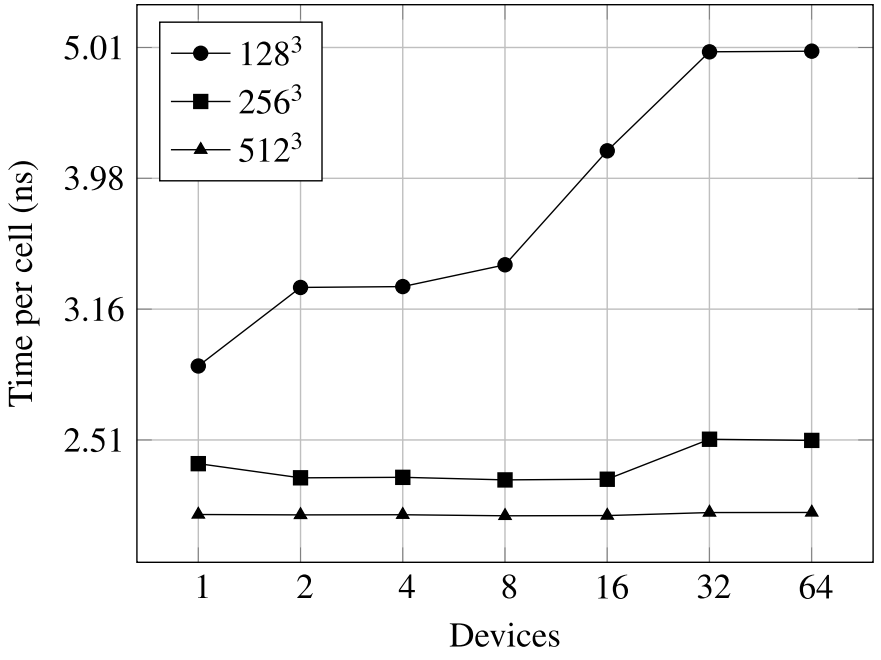


Performance

Strong scaling



Weak scaling



Big Picture

- Compute is cheap, bandwidth is expensive
- The gap between arithmetic performance and memory bandwidth continues to widen on future hardware
- Need to do everything we can to reduce data transfers

Highlights

- Maximizing intra-node data locality is key
- Z-order curve a simple and efficient strategy for processor assignment
- Parallel communication and compute
- Pipelined packing, transfer, unpacking

Performance

- In line with the expected theoretical scaling
  - 50-87% scaling efficiency from one to 64 GPUs
- Roughly 50 to 60x per-node speedup compared to multi-core CPU runs

Performance

- Systems
  - 4x NVIDIA V100 GPUs
  - 2x 64-core AMD Rome 7H12 64-core CPUs (Mahti)
  - 2x 20-core Intel Xeon Gold 623 CPUs (Puhti)
- Speedups compared to a multi-core CPU solver
  - 18x and 53x improvement per node
  - 20x and 60x improvement on 16 nodes
- Energy efficiency (perf. per Watt)
  - 9x and 12x improvement