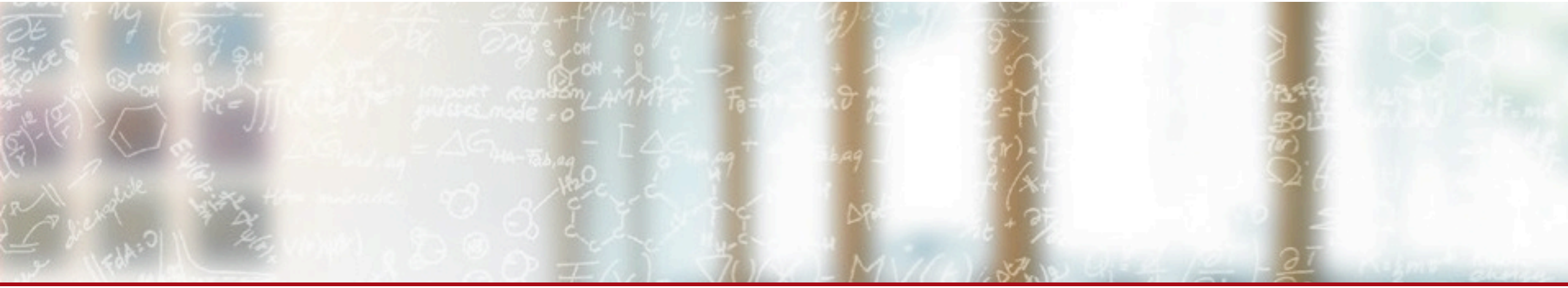
CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

ETH zürich



Part 1: Porting and Supporting GPUs at CSCS

JSC GPU Seminar Series

Ben Cumming, CSCS

January 28, 2020

Introduction: CSCS and SSL

CSCS: Swiss National Supercomputing Center

- Headquarters and machine room located in Lugano
- Offices in at ETHZ in Zürich

CSCS is accessible for scientists around the world via open, peer-reviewed calls:

- ‘small’ proposals (<1M node hours) → via national call
- ‘large’ proposals (>1M node hours) → via PRACE

GPUs in production at CSCS

- CSCS was an early adopter of GPU technology
  - Motivated by limits to CPU scaling in the face of power limitations and market forces (video games!)
- 2011: **Tödi**
  - Cray XK7 (NV-K20 GPU)
  - testing and development.
- 2013: **Piz Daint**
  - Cray XC30
  - 5000+ nodes NV-K20X GPU
  - #3 TOP 500, #1 Green 500
- 2016: **Piz Daint** upgrade
  - Cray XC50
  - 5700 Nodes NV-P100 GPU
  - #6 TOP 500 today

- 2016: **Kesch**
  - Cray CS-Storm Cluster
  - Nodes with 8 K80 GPUs
  - First national production weather system to use GPUs

The start of a larger trend…

GPUs will power ever increasing proportion of the top capability systems

- With the ratio of GPUs to CPU sockets increasing!
  - Summit [3:1]
  - Cray Shasta [4:1]
    - CSCS successor to Daint
    - Perlmutter
    - Aurora
    - Frontier

    } DOE Pre-Exascale and Exascale systems
  - Marconi100 @ Cineca [2:1]
  - JUWELS GPU Booster @ JSC [2:1]
  - EuroHPC Pre-Exascale ~150 PetaFLOP systems (Cineca, BSC, CSC) [?:1]

- Applications that don't run on GPUs won't run at scale in the near future…
  - … and risk becoming irrelevant

Practical FLOPs

- HPL results don't necessarily translate to real-world applications
  - Many early users of Piz Daint didn't see benefits without GPU-enabled applications
- Getting ~100% of users running on GPUs doesn't happen overnight
  - Porting legacy codes is… often really hard
  - Developing new codes with GPU performance and features of legacy apps is… daunting
- CSCS needed to work with application developers, users and vendors
  - There is no one size fits all approach for every code, domain or community
- CSCS needed to invest in software development to:
  - Target library development in areas where we can have the biggest impact for key communities
  - Encourage adoption of up to date development practices

SSL Group: 30 Developers For Portable Scientific Applications


Nora Abi Akar


Anton Afanasyev


Marco Bettiol


Mauro Bianco


John Biddiscombe


Fabian Bösch


Benjamin Cumming


Lukas Drescher


Nur Aiman Fadel


Andreas Fink


Simon Frasch


Daniel Ganellari


Enrique González


Linus Groner


Alberto Invernizzi


Shoshana Jakobovits


Marko Kabic


Prashanth Kanduri


Anton Kozhevnikov


Teodor Nikolov


Simon Pintarelli

Auriane Reverdell


Mikael Simberg


Raffaele Solcà


Mathieu Taillefumier


Felix Thaler

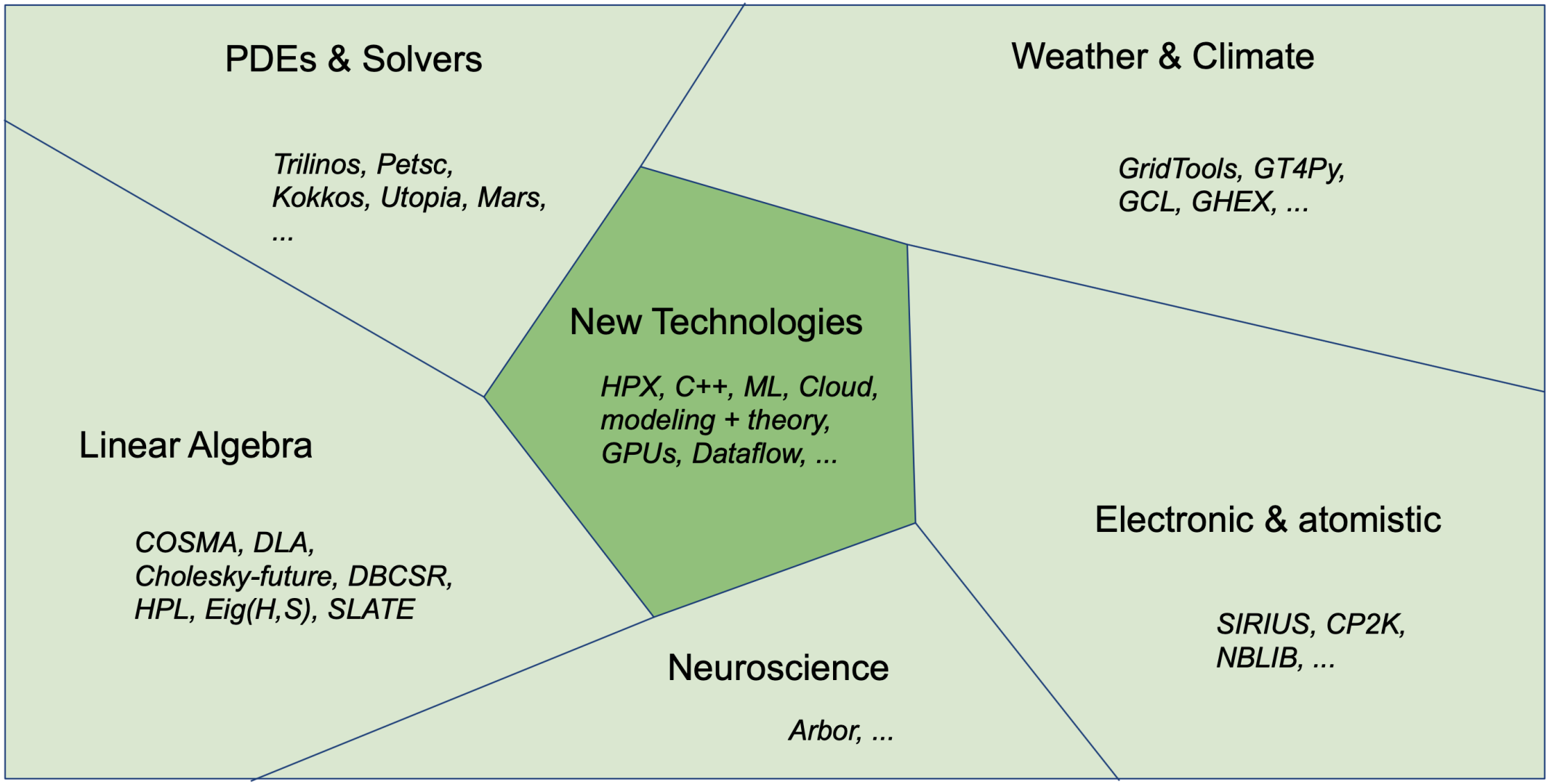

Joost VandeVondele


Stefan Velja


Hannes Vogt


Sam Yates

SSL Projects



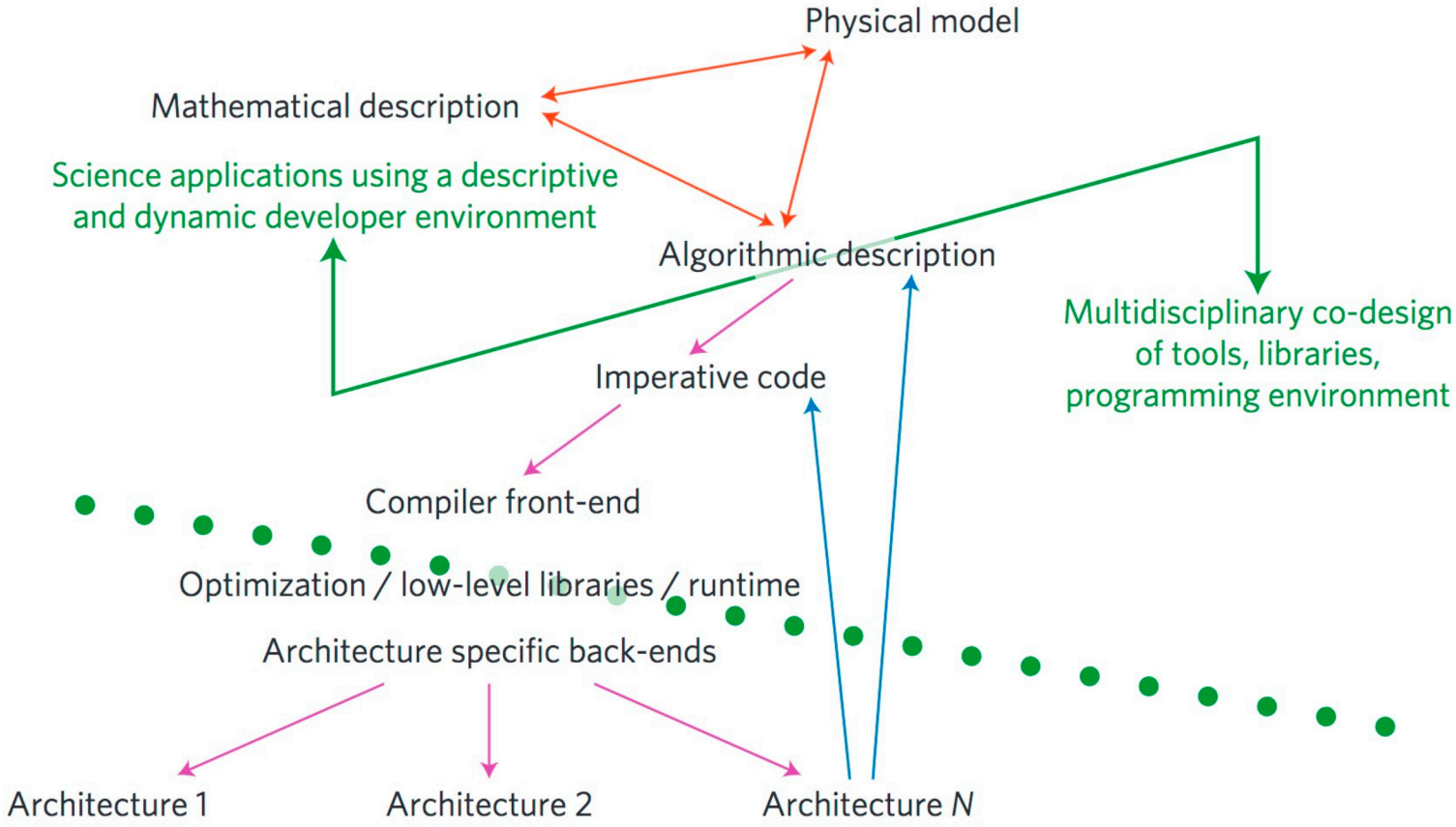
Porting To GPU: Diving In

- Analyze application
  - Which parts to move to GPU?
  - Which parts to keep on CPU?

1. Allocate memory on GPU or CPU
   - malloc → cudaMalloc
   - OpenACC/MP directives
2. Port the for loops
   - CUDA kernels
   - OpenACC/MP directives
   - Library calls (cuBLAS, cuFFT)
3. Manual memory movement
   - cudaMemcpy
   - OpenACC/MP directives

Now we have two applications
- What if we want to add support for another architecture?
  - OpenACC/OpenMP will diverge from original version
- Two applications are harder to maintain than one

Portable Approach: Separation of Concerns



Separate software in performance portable backends and rapid prototyping frontends. Requires a change in how the community writes and maintains software

Thomas C. Schulthess

Nature Physics, 11 (5): 369-373, London: Nature Publ. Group, 2015.

CUDA is less than 3% of GPU-enabled SSL codebases

Arbor

969 commits | 9 branches | 0 packages | 4 releases | 14 contributors | View license

C++ 93.2% | Python 2.3% | Cuda 1.8% | CMake 1.4% | AMPL 0.5% | Julia 0.4% | Other 0.4%

DBCSR

2,417 commits | 6 branches | 0 packages | 24 releases | 14 contributors | GPL-2.0

Fortran 81.8% | Python 8.4% | C++ 3.6% | CMake 2.7% | Shell 1.2% | Makefile 0.7% | Other 1.6%

Sirius

7,012 commits | 3 branches | 0 packages | 30 releases | 10 contributors | BSD-2-Clause

C++ 82.9% | Python 10.2% | Fortran 2.3% | CMake 2.2% | Cuda 1.7% | C 0.5% | Shell 0.2%

GridTools

5,965 commits | 12 branches | 0 packages | 13 releases | 17 contributors | View license

C++ 90.0% | CMake 3.2% | Cuda 2.6% | Python 2.2% | Shell 1.0% | Fortran 0.8% | Other 0.2%

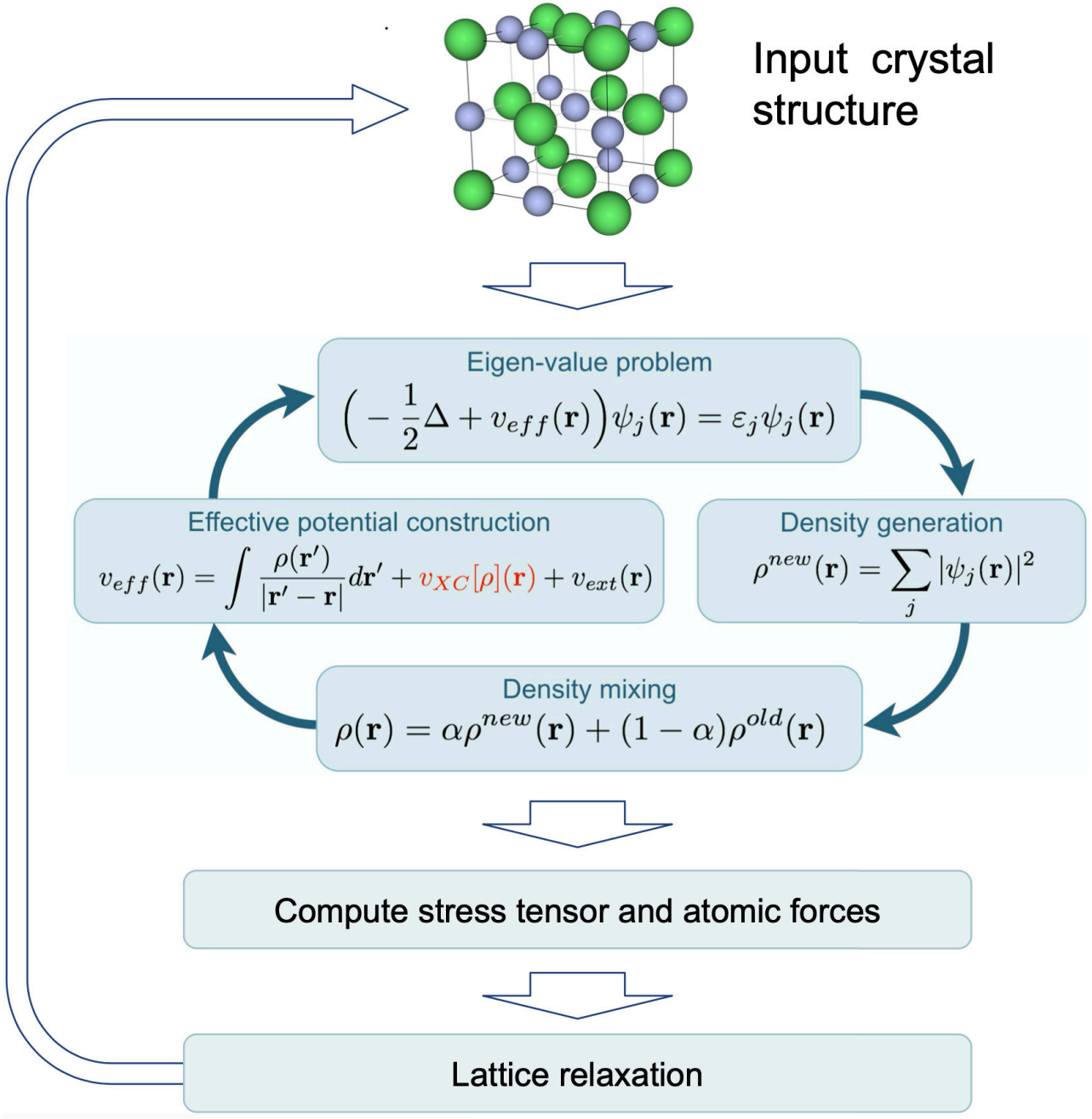
Less than or equivalent to CMake…

- There is hope if we separate low-level backends

Case Study: Sirius

Sirius

- QuantumEspresso (QE) is a key application for CSCS users
  - *Ab initio* quantum chemistry methods of electronic-structure calculation and materials modeling
  - Based on DFT methods
  - Written in F90 with MPI and OpenMP
  - Heavy use of BLAS, LAPACK and FFT libraries

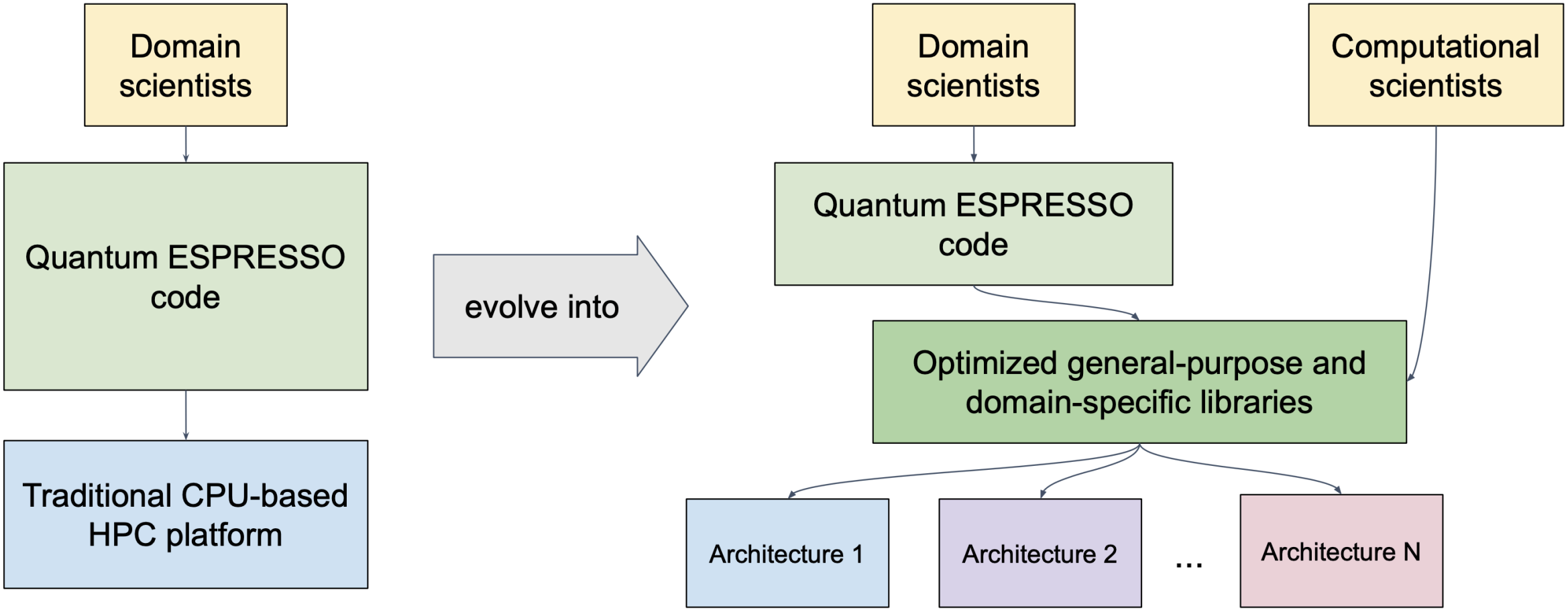


Options for adding GPU support

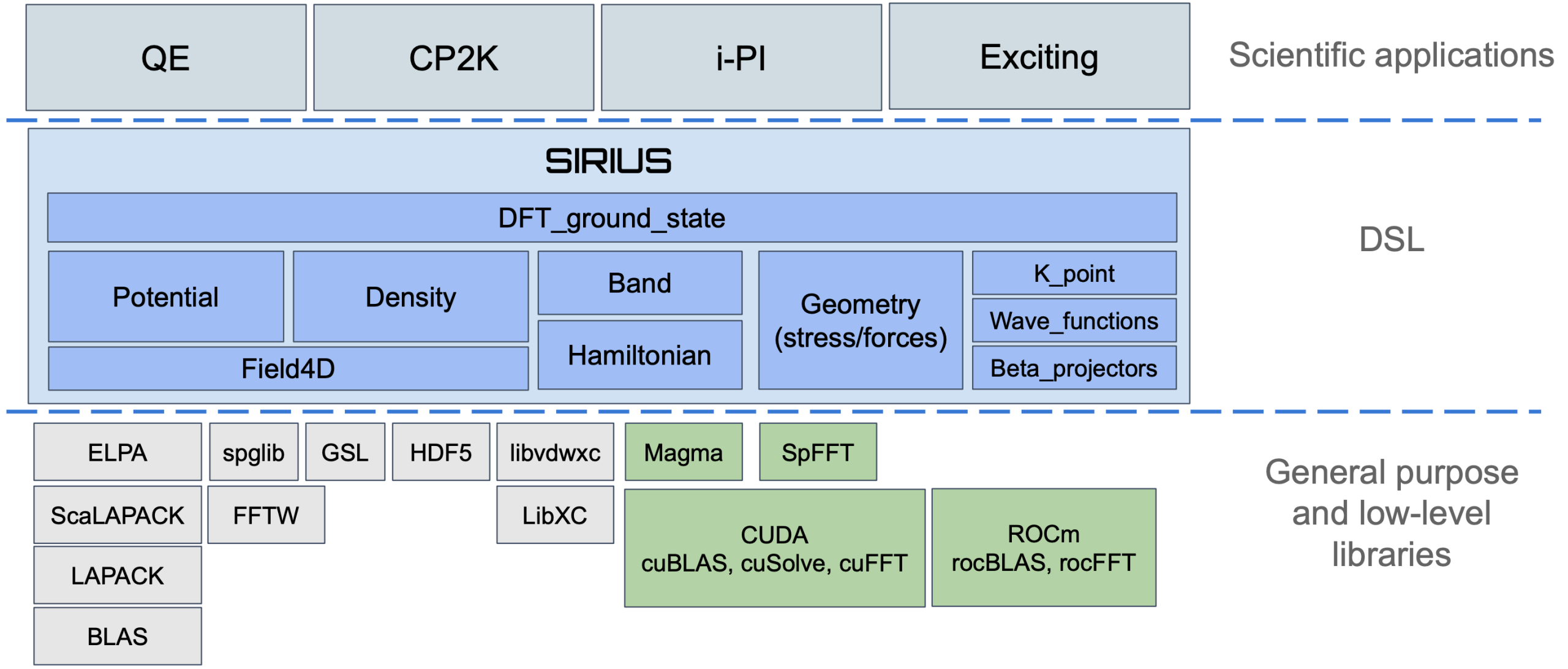
- Intercept calls to BLAS/LAPACK/FFT (e.g. MKL) and substitute with GPU-accelerated equivalents (e.g. CUBLAS)
  - No performance gain due to high memory transfer overheads
- Work directly on Fortran code: OpenACC or OpenMP 4.5
  - Requires large refactoring of QE (and buy in from developers)
  - OpenACC has tenuous vendor support outside of NVIDIA
  - OpenMP is still unproven in production for accelerators
  - Directives make code more complex, difficult to reason about, and rely on compiler magic.
  - It is very difficult to hire ambitious young developers to work on Fortran.
- Rewrite basic DFT building blocks in a stand-alone library
  - Requires buy in from developers

SIRIUS library: separation of responsibilities

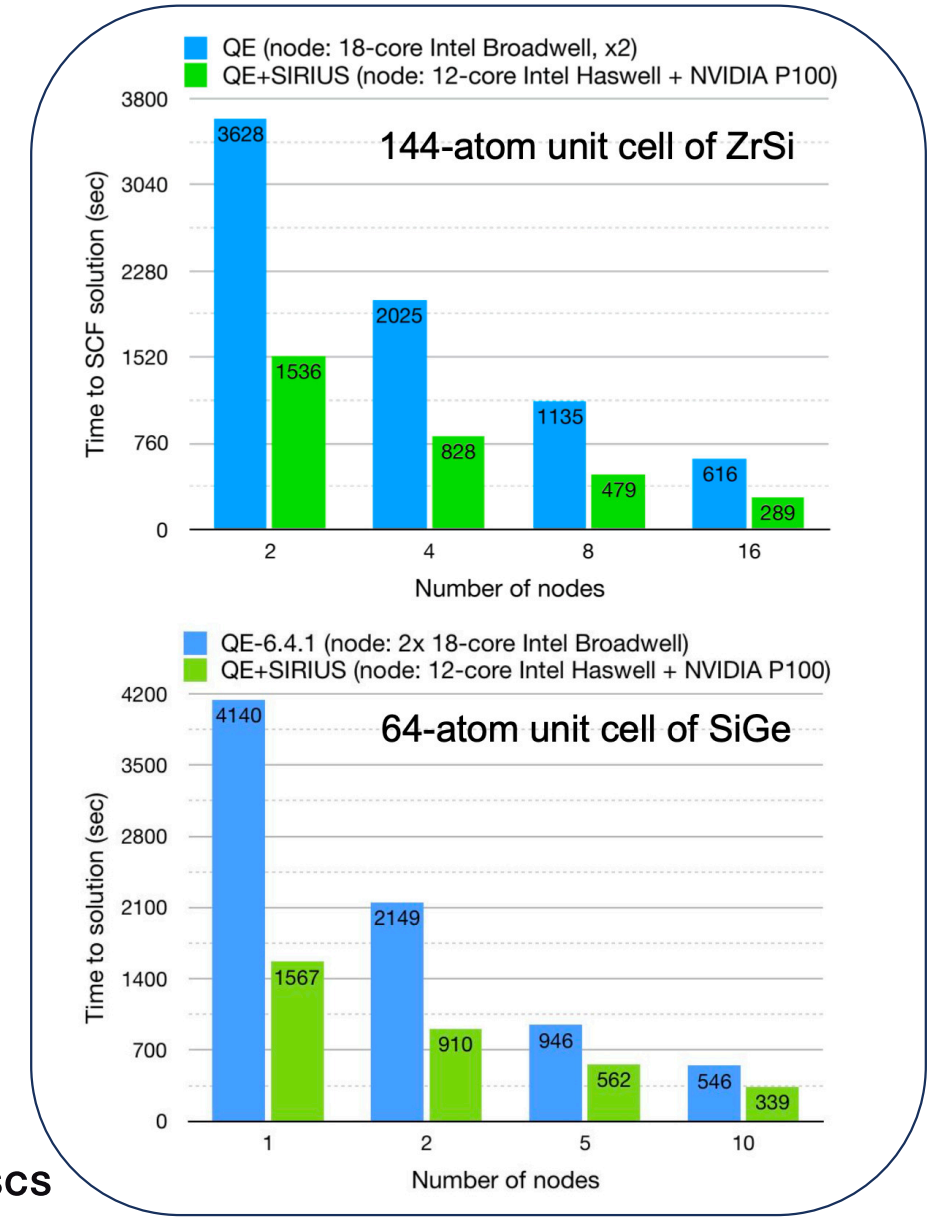
- Library that presents abstraction of the algorithms and methods (describe what, not how)
- Calling code does not interact directly with hardware back ends

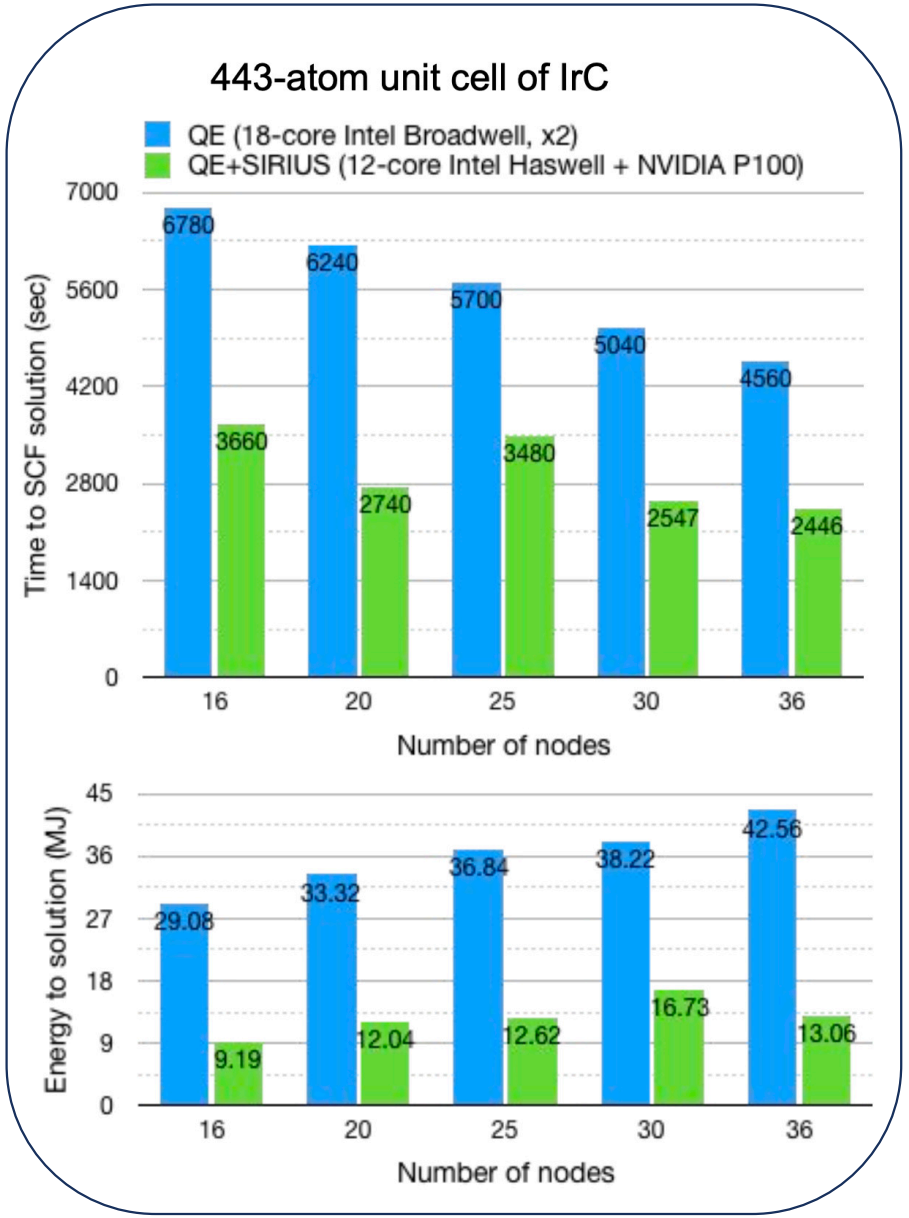


Current software stack: not just QE



Performance Benchmarking of Sirius





Community

- CSCS maintains a version of QE-Sirius that is always up to date with Master
  - Open source on GitHub
    - https://github.com/electronic-structure/SIRIUS
    - https://github.com/electronic-structure/q-e-sirius
  - Support for
    - Intel vectorized CPUs
    - NVIDIA GPUs (CUDA)
    - AMD GPUs (HIP/ROCM)
  - Available to users on Piz Daint as module
    - https://user.cscs.ch/computing/applications/sirius/
- QE developers maintain a CUDA Fortran version of CP2K
  - Good performance
  - Less features than QE-Sirius
  - Not portable to non-NVIDIA architectures

Case Study: DBCSR

DBCSR: Sparse Matrix Multiplication Library

DBCSR is:

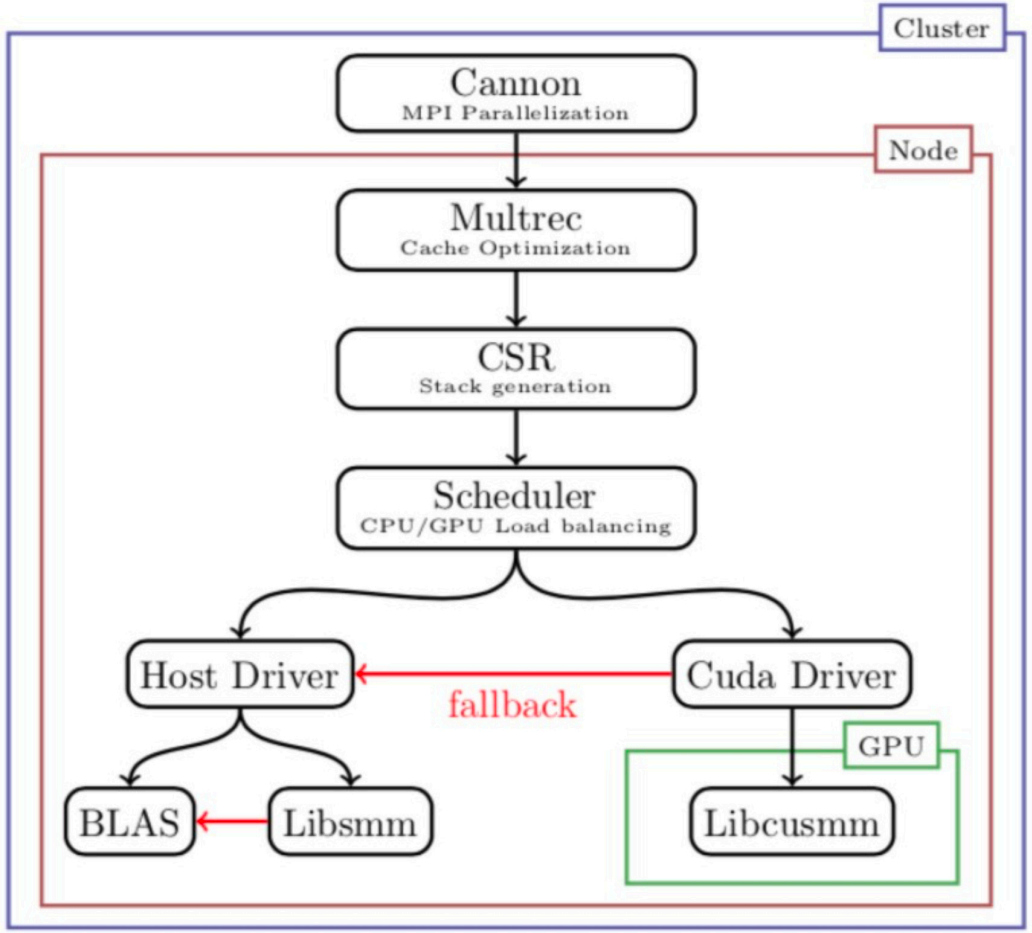
- Distributed sparse matrix-matrix multiplication
  - Sparsity pattern of small dense blocks
- MPI and OpenMP for distribution and scheduling
- Sparse linear algebra backend for CP2K (quantum chemistry)

DBCSR's GPU back end:

- Batched small dense block multiplication
- Highly parameterized matrix-matrix CUDA kernels

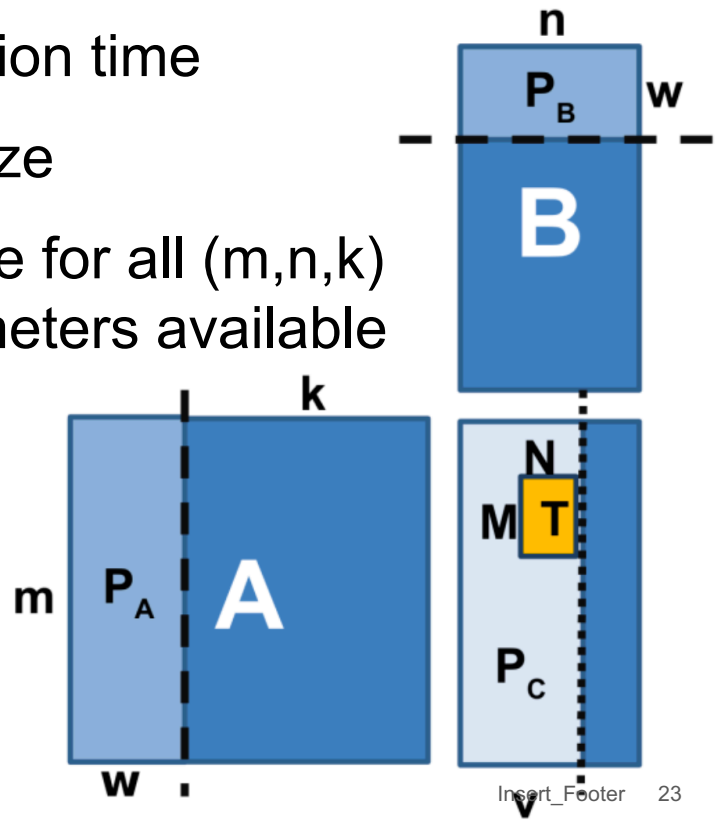
DBCSR

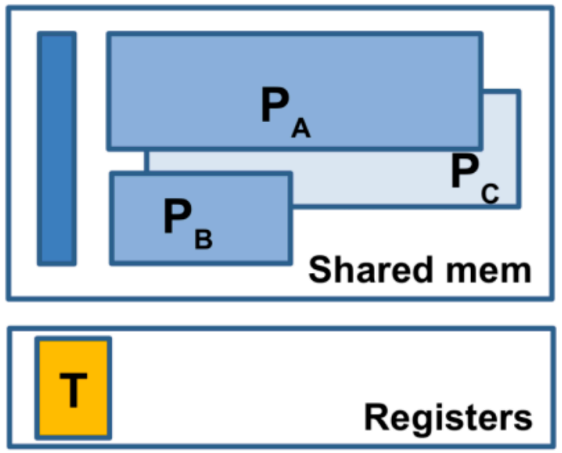
- DBCSR was implemented by CP2K developers with a library API
  - Development of backends has minimal impact on CP2K calling code
- Library interface allowed one developer, *Shoshana Jakobowits*, to perform the porting work described here
- Originally a sub set of autotuned kernels were pre-compiled into the library.
  - Fall back to CPU if unavailable

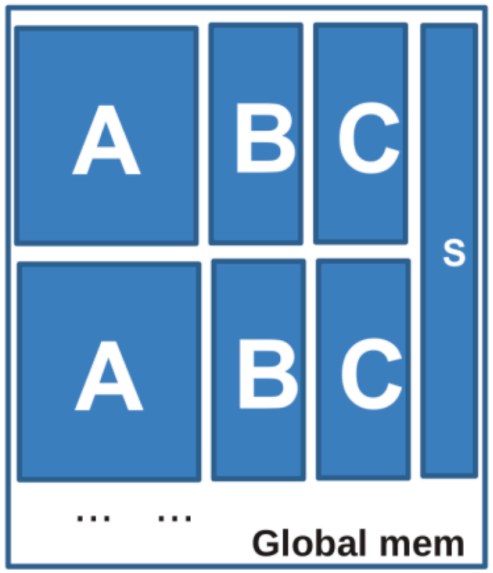


DBCSR Step 1: Just in time (JIT) compilation

- A single (m,n,k) kernel has many parameters to tune on GPU.
  - Autotune the parameters ahead of time, and store in library
- JIT only the required kernels at run time
  - Make a best guess when autotuned parameters unavailable.
- Reduced compilation time
- Reduces binary size
- Can generate code for all (m,n,k) sets if good parameters available

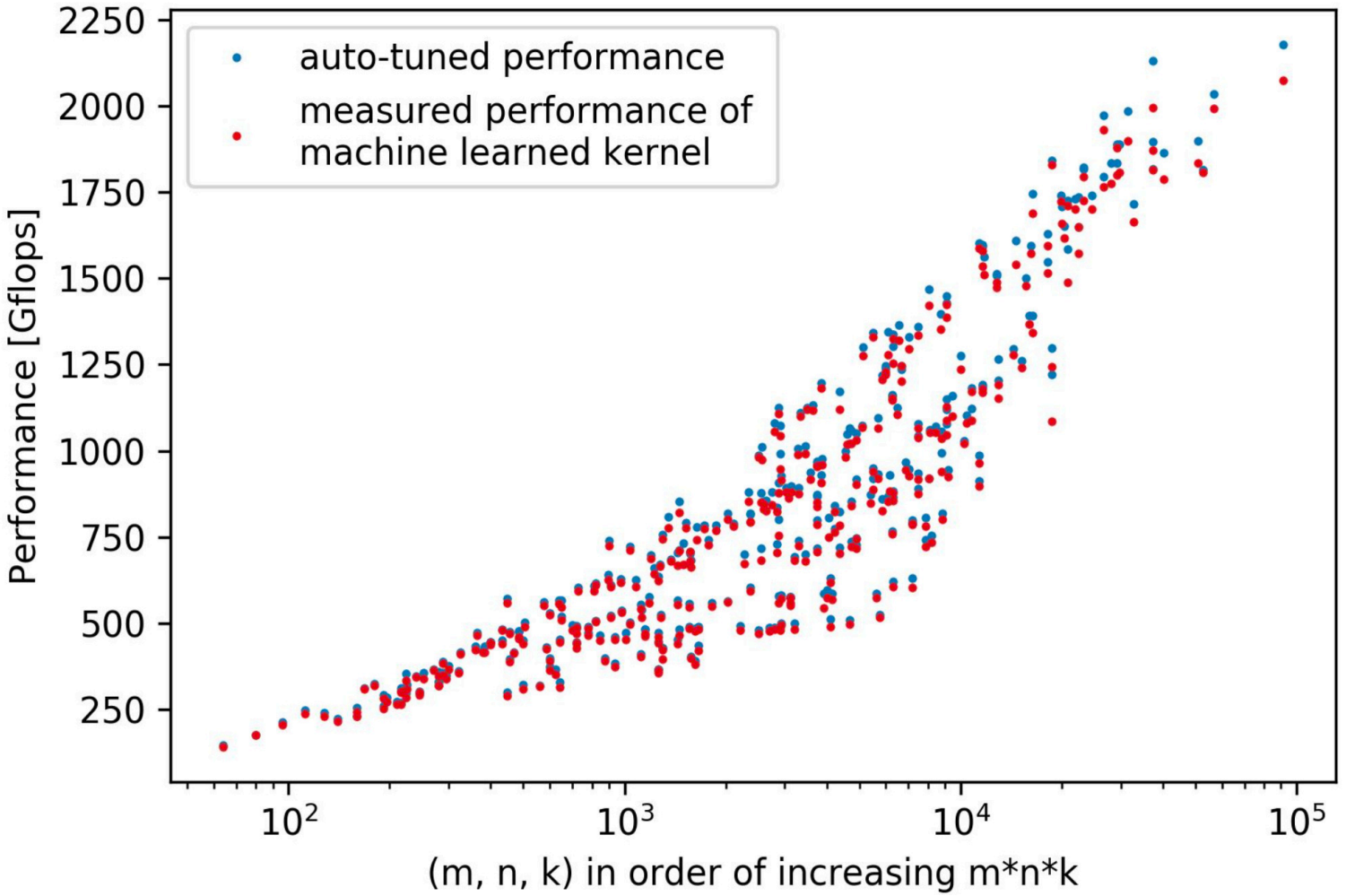






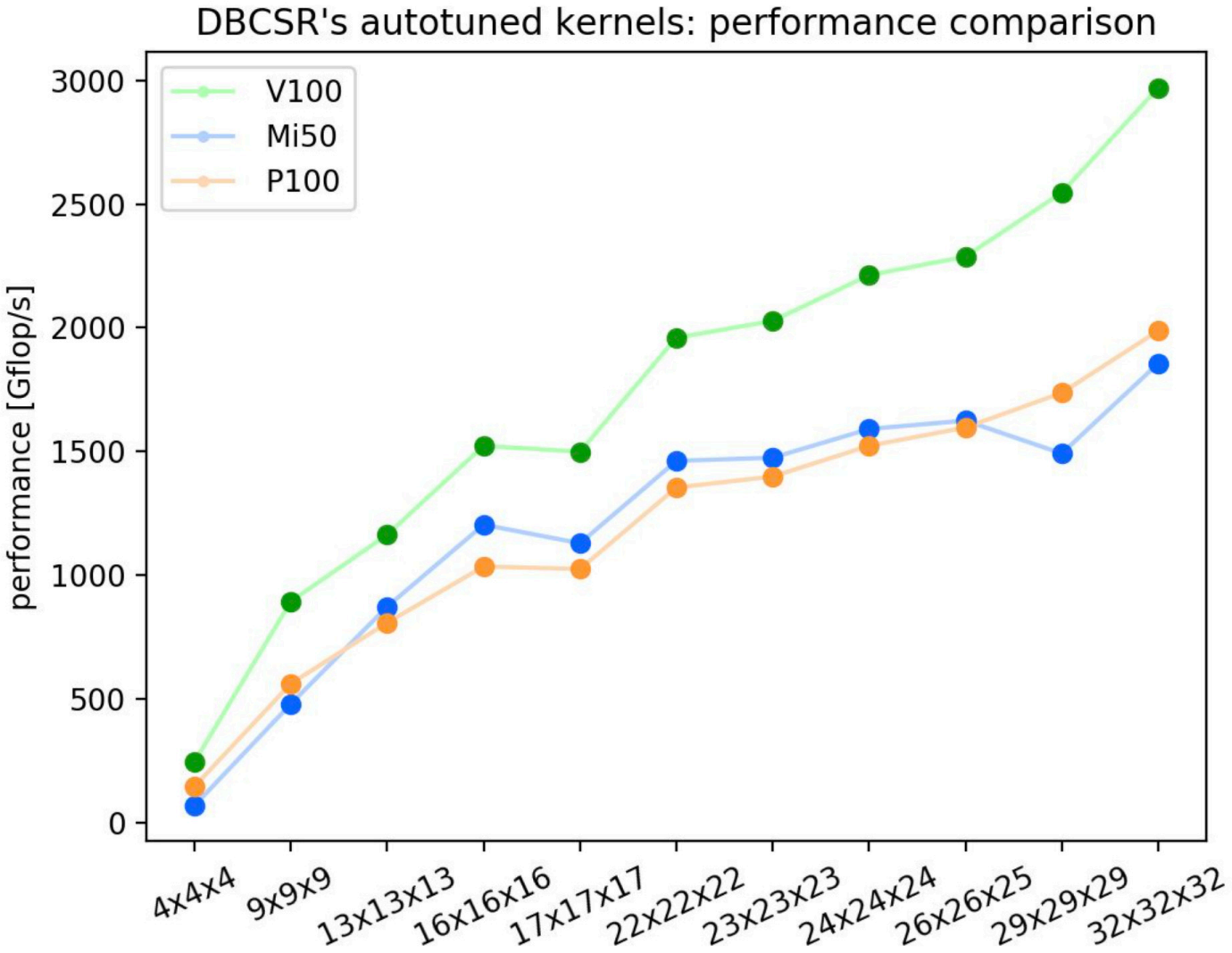
DBCSR: machine learning to determine JIT parameters

- Derive performance model from a sub-set of auto-tuning data that can predict performance over entire (m,n,k) space
- Use boosted regression trees
  - Requires significant GPU resources, but far less than autotuning.
- Predicted kernels are within 3% of autotuned kernels on average.



DBCSR: machine learning portability

- The cost of porting to new architectures is greatly reduced
  - resources required to generate inputs for new GPU are much lower
  - Also applies to new generations of NVIDIA GPU



DBCSR: AMD GPUs

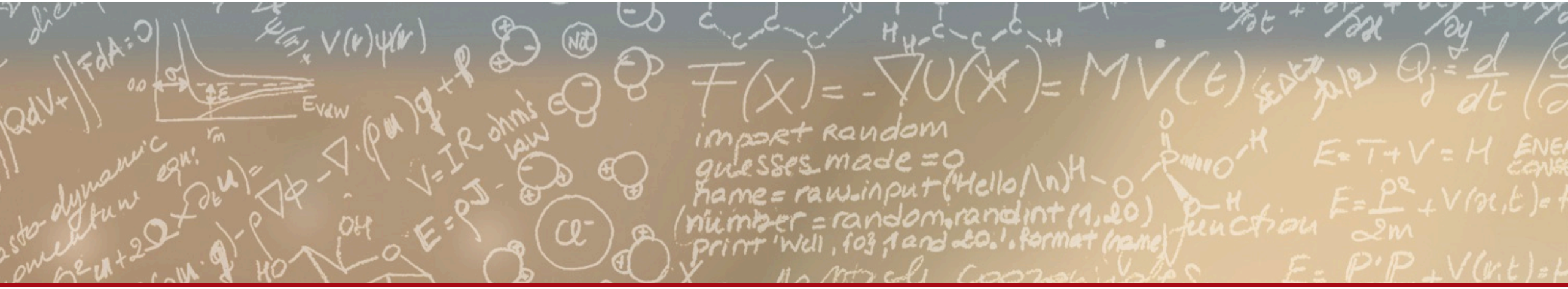
- … wait, the last slide showed results for AMD Mi50 GPUs
- The CUDA back end took years to develop
  - The up from cost of library design required to add the first new back end.
- The AMD HIP back end took 21 work days to write
  - **Code generation** means that there are no CUDA kernels to port
  - Wrapping HIP API calls with macros: `cudaMalloc` → `hipMalloc`
  - No code logic duplicated.

- Large initial investment in developing DBCSR by CP2K developers
  - As a result CP2K can be adapted to new accelerator platforms more easily
  - CP2K will still be running on the biggest systems in 10 years time
  - … going strong for a legacy Fortran application!

Conclusion

Before diving into porting to GPU

- Make the up front payment to refactor the code to use a domain-specific library interface used by the front end code
- Interface uses domain-specific and algorithmic language relevant to application



And now, for a presentation I prepared earlier…

CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

arbor

Arbor – morphologically-detailed neural network simulation on GPUs

JSC GPU Seminar Series

Nora Abi Akar, Ben Cumming, Vasileios Karakasis, Anne Küsters, Wouter Klijn, Alexander Peyser, Stuart Yates

January 28, 2020

Human Brain Project

This research has received funding from the European Union's Horizon 2020 Framework Programme for Research and Innovation under the Specific Grant Agreement No. 720270 (Human Brain Project SGA1), and Specific Grant Agreement No. 785907 (Human Brain Project SGA2).

CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

`Arbor` and its aims

`Arbor` is library for simulation of morphologically-detailed cells in large networks on HPC systems.

- the biggest users of HPC resources in HBP.
- `Arbor` is being developed as part of HBP.
- **key aim**: open research infrastructure for neuroscience.
- **key aim**: enabling neuroscience on all HPC systems.

Requires a **rich interface** for defining models.

- Simulation of electrical current in arbitrarily complex morphologies.
- Arbitrary ion channel and synapse models.
- Inter-cell communication via spikes on arbitrary networks.

CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

ETH zürich

User API: Separation of concerns

Step 1: Model abstraction (which model)

User models are described by a **recipe**, which take a cell number and give:

- a description of the cell
  - piecewise linear morphology
  - named regions and locations
  - ion channel and synapses
- spike targets
- spike sources
- network connections that terminate on the cell

Recipe descriptions are functional, with lazy evaluation for efficient parallel model construction.

Recipes contain no hardware or implementation details.

Step 2: Hardware context (which hardware)

Select hardware resources

```
import arbor
from mpi4py import MPI

rec = my_recipe() # user defined model
ctx = arbor.context(threads=12, gpu_id=0, mpi=MPI.COMM_WORLD)
```

Users can select hardware resources at run time:

- Number of threads in thread pool
- Which GPU [optional]
- Which MPI communicator [optional]

Step 3: Instantiate model

Instantiate model on target compute resources

```
import arbor
from mpi4py import MPI

rec = my_recipe() # user defined model
ctx = arbor.context(threads=12, gpu_id=0, mpi=MPI.COMM_WORLD)
sim = arbor.simulation(rec, ctx)
```

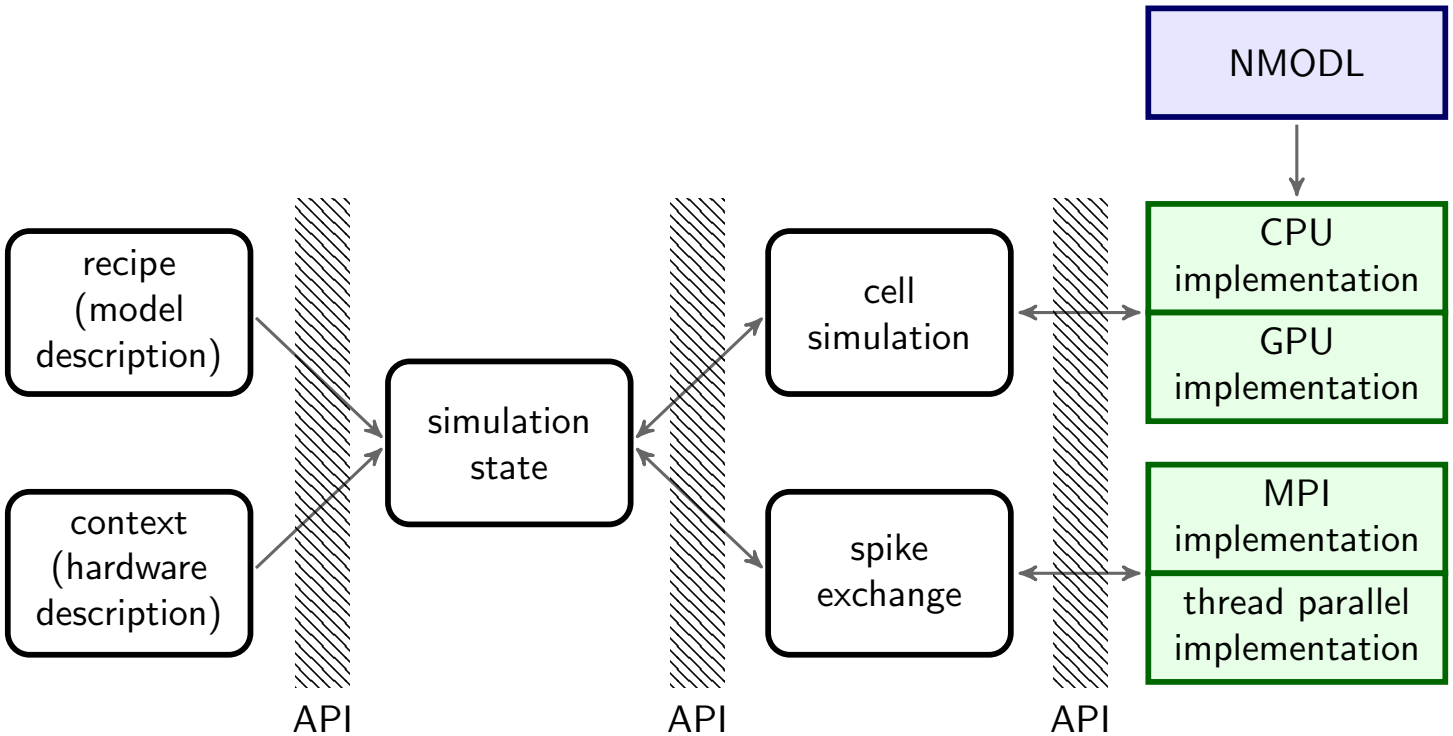
A simulation object:

- Instantiates target-specific data structures and call backs
- Provides a generic interface for:
  - steering simulation
  - sampling spikes, voltages, etc.
- Has no global state
  - multiple simulations can be instantiated simultaneously.

Caller can optionally provide hints on how to assign model to hardware resources.

Separation of concerns

Components communicate via APIs allow that allow implementation of new cell models, communication methods, hardware back ends etc.



CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

ETH zürich

Target-specific code generation

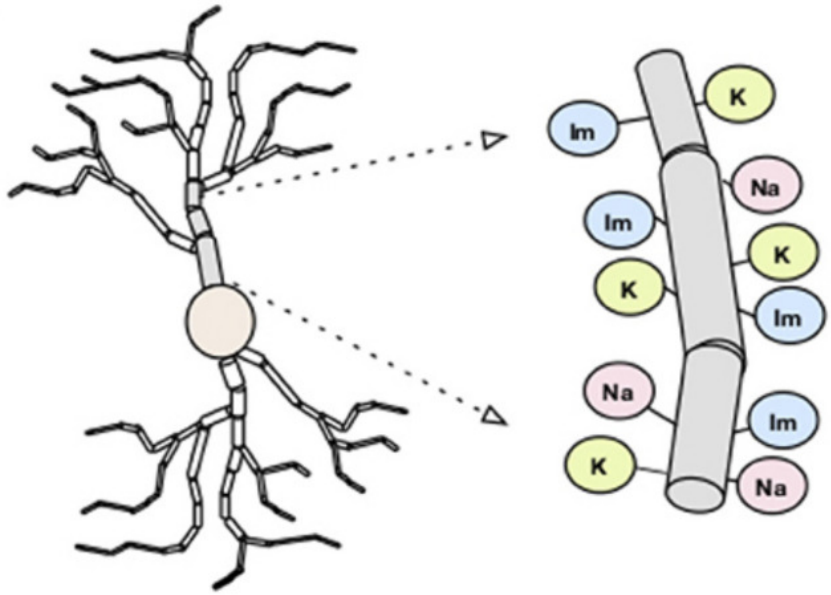
Cell models are complicated

Drawing of a Purkinje cell in the cerebellar cortex by Santiago Ramòn y Caja.

Compartmentalization

Cables are broken into individual **compartments**, then ion channels and synapses (mechanisms) are assigned to compartments.

- Each mechanism has multiple ODEs per compartment.
- Fine-grained parallelism (SIMD + SIMT) from one compartment per lane for current and ODE state update.
- Pack multiple cells together to expose more SIMT parallelism.



Pramod Kumbhar et. al. *CoreNEURON : An Optimized Compute Engine for the NEURON Simulator*, Frontiers in Neuroinformatics, 2019.

Efficient dynamics

Two problems:

- Efficient ion channel and synapse (mechanism) computation requires hardware-specific implementations.
- Extensibility demands that users can define their own mechanisms.

It is not practical to explicitly implement optimized code for each possible mechanism on each extant platform.

Solution: use a DSL to describe mathematics, and generate optimized code from that for each platform.

NMODL language

NMODL is a domain specific language for describing the dynamics of ion channel and synapses inside compartments:

- state and update rules (ODEs).
- contribution to membrane current and ionic transport.

Synapse with exponential conductance decay

```
NEURON {
    POINT_PROCESS expsyn
    RANGE tau, e
    NONSPECIFIC_CURRENT i
}
PARAMETER {
    tau = 2.0 (ms)
    e = 0   (mV)
}
STATE { g }
INITIAL { g=0 }
BREAKPOINT {
    SOLVE state METHOD cnexp
    i = g*(v - e)
}
DERIVATIVE state { g' = -g/tau }
NET_RECEIVE(weight) { g = g + weight }
```

CUDA for GPUs

We chose CUDA for the GPU back end:

- Abor is a C++ library, and CUDA interfaces very well with C++ (it is a superset of C++).
- There was no domain-specific library for Arbor's motifs (e.g. BLAS).
- Arbor developers had strong background in CUDA development.
- CUDA enforces a GPU-specific programming model.
  - OpenCL and HIP/ROCM programming models for GPUs are almost one to one compatible with CUDA.
- CUDA has strong support outside HPC (e.g. ML and AI).

CUDA kernel generation

The modcc compiler generates for each target:

- target-specific kernels for state update, current contribution, event handling etc.
- type-erased C++ wrapper for calling from simulation.

CUDA kernel for expsyn state integration

```
__global__
void nrn_state(mechanism_gpu_expsyn_pp_ params_) {
    int n_ = params_.width_;
    int tid_ = threadIdx.x + blockDim.x*blockIdx.x;
    if (tid_<n_) {
        auto node_index_i_ = params_.node_index_[tid_];
        value_type dt = params_.vec_dt_[node_index_i_];
        value_type a_0_, ll0_, ll1_;
        a_0_ =  -1/params_.tau[tid_];
        ll0_ = a_0_*dt;
        ll1_ = ( 1+ 0.5*ll0_)/( 1- 0.5*ll0_);
        params_.g[tid_] = params_.g[tid_]*ll1_;
    }
}
```

Target-specific optimizations

Arbor's NMODL compiler, modcc, can use algorithms that are optimized for a specific target.

For example, the current update in expsyn `i = g*(v-e)`:

- Synapses on the same compartment contribute to the same per-compartment current
- There can be 10-10'000 synapses on a single compartment.
- Race condition if when multiple threads add to the same value.

CUDA atomic operations don't scale well enough

- Arbor uses an optimized **reduce by key algorithm**

Reduce by key

Vanilla C++ reduce by key

```
void reduce_by_key(const double *values, const int *index,
                   int n, double* out)
{
    for (int i=0; i<n; ++i) {
        out[index[i]] += values[i];
    }
}
```

Note: this is also non-trivial to implement in the vectorized multicore back end.

Note: `index` is monotonically increasing.

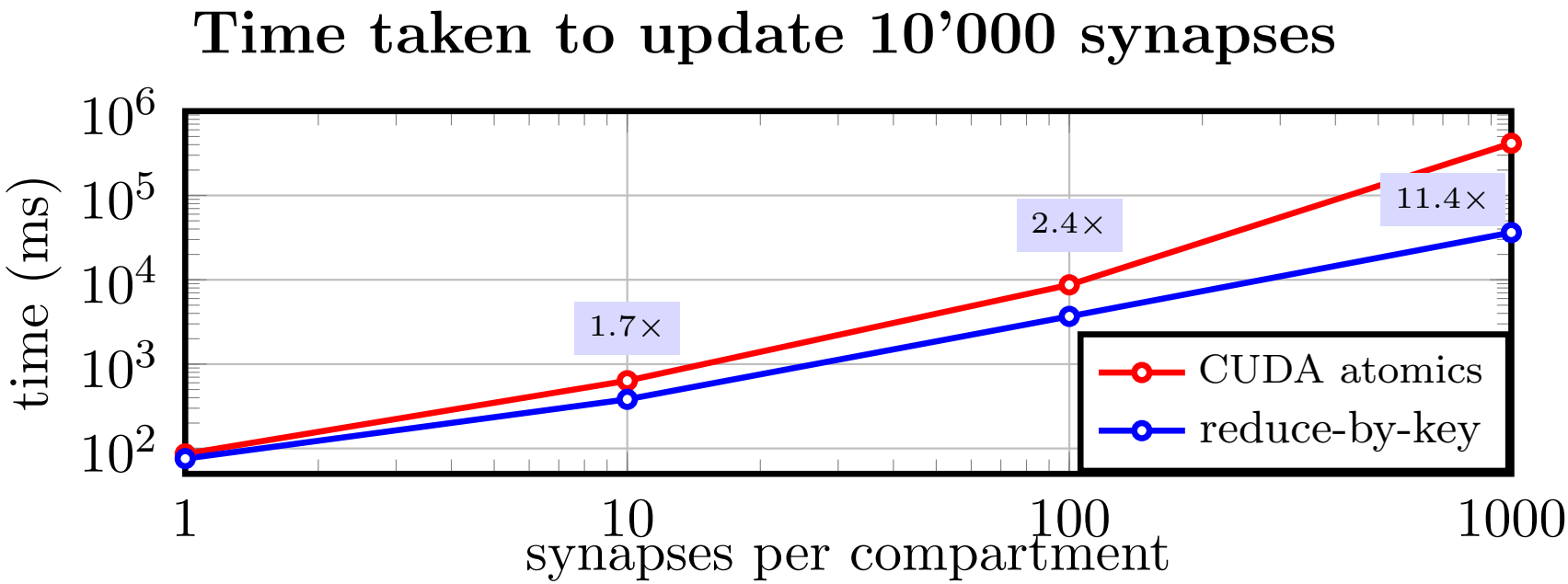
Reduce by key

CUDA kernel for expsyn current update

```
__global__ void nrn_current(mechanism_gpu_expsyn_pp_ P_) {
    int n_ = P_.width_;
    int tid_ = threadIdx.x + blockDim.x*blockIdx.x;
    unsigned lane_mask_ = __ballot_sync(0xffffffff, tid_<
        n_);
    if (tid_<n_) {
        auto idx_ = P_.node_index_[tid_];
        value_type conductivity_ = 0;
        value_type v = P_.vec_v_[idx_];
        value_type current_ = 0;
        value_type i = 0;
        i = P_.g[tid_]*(v-P_.e[tid_]);
        // atomicAdd(P_.weight_[tid]*i, P_.vec_i_+idx_);
        gpu::reduce_by_key(P_.weight_[tid_]*i, P_.vec_i_,
            idx_, lane_mask_);
    }
}
```

Reduce by key performance

Reduce by key implementation outperforms atomics by a factor of $2 - 10\times$ for more than a hundred synapses per compartment.



Adding new backends

Adding a backend for a new architecture is reasonably straightfoward. For example, adding a HIP backend for AMD GPUs:

1. Hand port CUDA kernels generated by modcc as POC.
2. Write a modcc printer to generate these kernels.
3. Write back end specific glue and helper code.
4. Iterate on exisiting unit and integration tests.

key lesson: Write unit tests.

CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

ETH zürich

GPU Performance

Benchmark Systems

Single node specs of Cray systems at CSCS.

	Daint-mc	Daint-gpu	Tave-knl
CPU	Broadwell	Haswell	KNL
memory	64 GB	32 GB	96 GB
CPU sockets	2	1	1
cores/socket	18	12	64
threads/core	2	2	4
vectorization	AVX2	AVX2	AVX512
accelerator	–	P100 GPU	–
interconnect	Aries	Aries	Aries
MPI ranks	2	1	4
threads/rank	36	24	64
configuration	–	CUDA 9.2	cache,quadrant
compiler	GCC 7.2.0	GCC 6.2.0	GCC 7.2.0

Single node scaling table

Ring network of cells with 10'000 synapses, 145 compartments and random morphologies.

	wall time (s)			energy (kJ)		
cells	`Arbor`-mc	`Arbor`-gpu	`Arbor`-knl	`Arbor`-mc	`Arbor`-gpu	`Arbor`-knl
32	**0.35**	**2.06**	**1.13**	**0.04**	0.25	0.17
64	**0.39**	**2.10**	**1.29**	**0.05**	0.25	0.22
128	**0.75**	**2.44**	**1.71**	**0.11**	0.33	0.34
256	**1.42**	**2.97**	**2.28**	**0.25**	0.43	0.55
512	**2.66**	**4.19**	3.36	**0.58**	0.67	0.97
1024	**5.12**	6.50	6.15	1.23	**1.13**	1.80
2048	**10.04**	11.11	12.27	2.53	**2.10**	3.63
4096	**19.93**	19.96	24.39	5.15	**3.95**	7.24
8192	39.66	**37.24**	48.65	10.37	**7.71**	14.45
16384	79.22	**71.65**	97.19	20.85	**15.10**	28.99

- Performance portable across architectures.
- Strong scaling higher on nodes with less parallelism.
- GPU is also energy efficient.

Conclusion

Retrospective

After 4 years of development we have come to realise that...

- Unit tests pay you back more the longer you have them.
- The amount of time spent hacking on low level GPU or vectorized code is much less than the time spent on interfaces.
  - But is less than the time spent hacking without interfaces.
- The multi-core vectorization backend ultimately took the same amount of effort as the CUDA backend.
- Amdahl's law matters on the GPU.

`Arbor` is under active, open, development.

`Arbor` is **open source software**:

`github.com/arbor-sim/arbor`