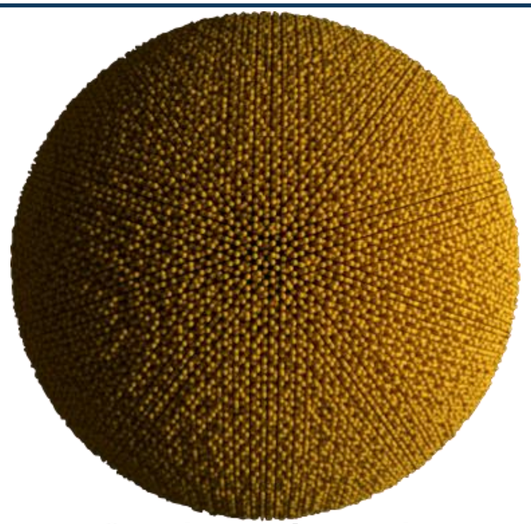
Auto-Tuning Kernel Launch Parameters for Maximum Performance

Joshua A. Anderson

Molecular dynamics

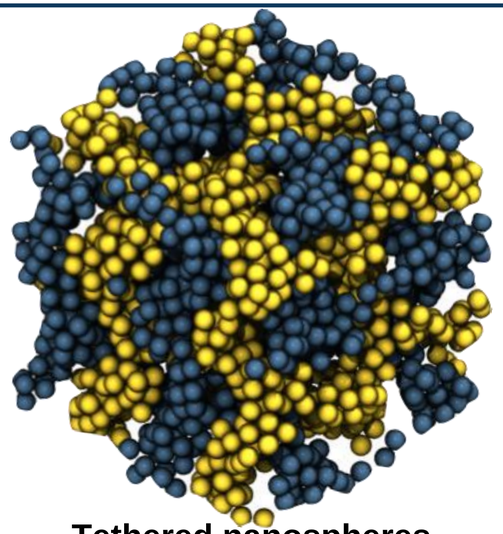
Monte Carlo



Quasicrystal growth

Molecular Dynamics

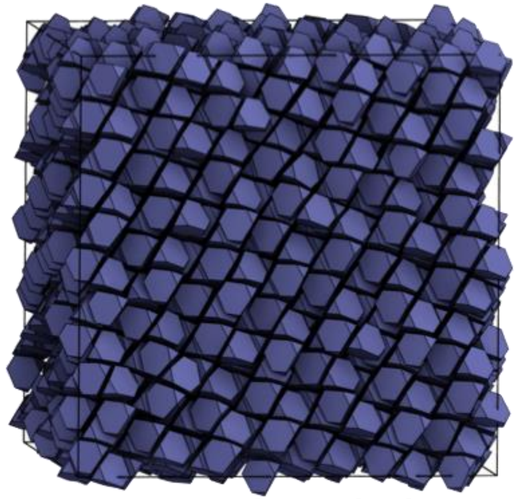
Engel M. et al., *Nature Materials* **14** 109-116, 2014



Tethered nanospheres

Langevin dynamics

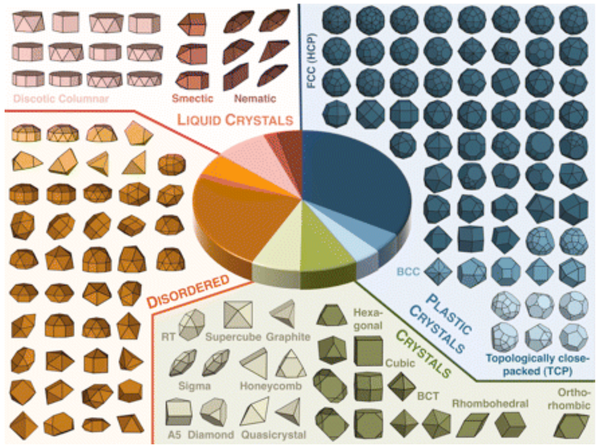
Marson, R, *Nano Letters* **14**, 4, 2014



Truncated Tetrahedra

Hard particle MC

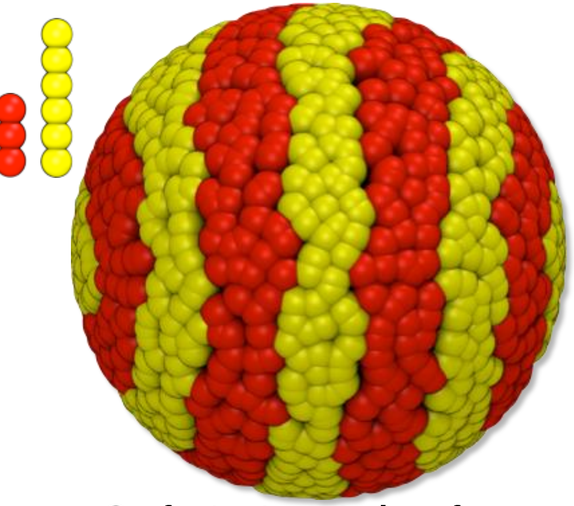
Damasceno, P. F. et al., *ACS Nano* **6**, 609 (2012)



Arbitrary polyhedra

Hard particle MC

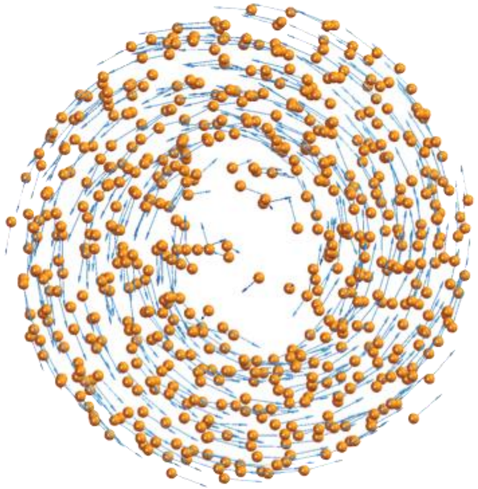
Damasceno, P. F. et al., *Science* **337**, 453 (2012)



Surfactant coated surfaces

Dissipative particle dynamics

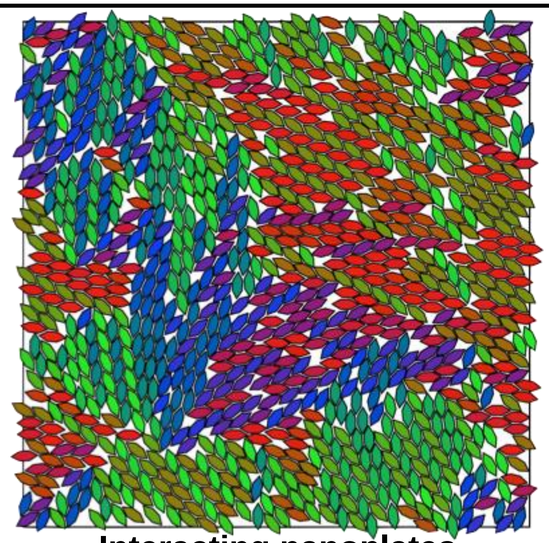
Pons-Siepermann, I. C., *Soft* matter **6** 3919 (2012)



Self-propelled colloids

Non-equilibrium MD

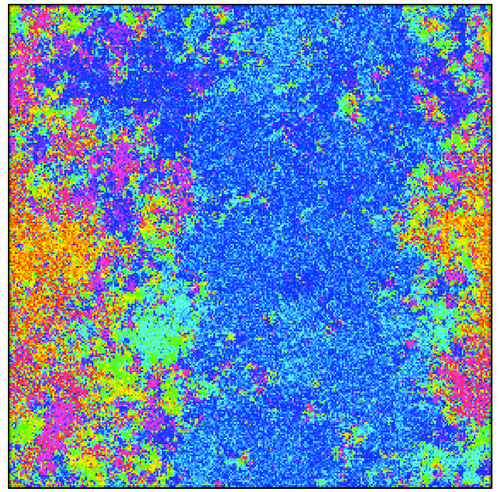
Nguyen N., *Phys Rev E* **86** 1, 2012



Interacting nanoplates

Hard particle MC with interactions

Ye X. et al., *Nature Chemistry* **cover article** (2013)



Hard disks - hexatic

Hard particle MC

Engel M. et al., PRE **87**, 042134 (2013)

Features in HOOMD-blue v1.0

Integration
- NVT (Nosé-Hoover)
- NPT
- NPH
- Brownian Dynamics
- Dissipative Particle Dynamics
- NVE
- FIRE energy minimization
- Rigid body dynamics

Snapshot formats
- MOL2
- DCD
- PDB
- XML

Simulation types
- 2D and 3D
- Triclinic box
- Replica exchange (via script)

Pair forces
- Lennard Jones
- Gaussian
- CGCMM
- Morse
- Table
- Yukawa
- PPPM electrostatics

Many-body forces
- EAM

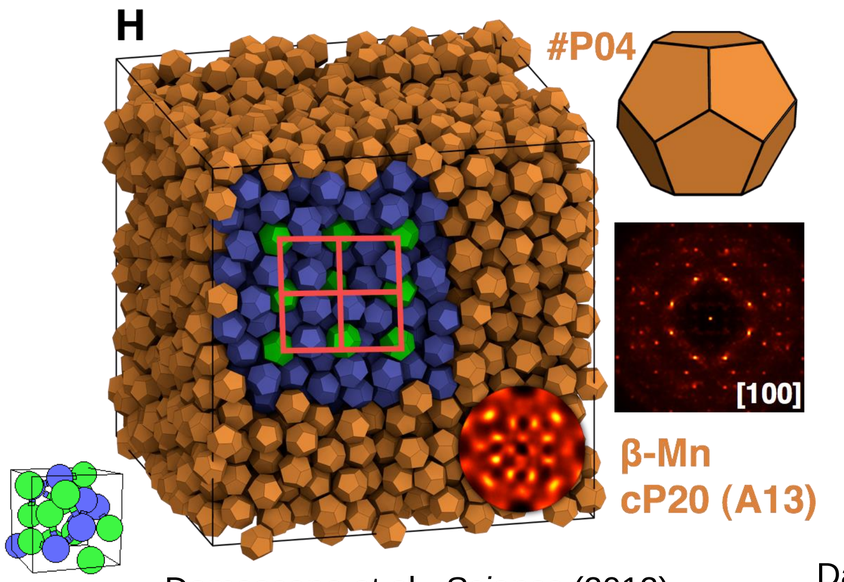
Bond forces
- Harmonic
- FENE
- Table

Angle forces
- Harmonic
- CGCMM
- Table

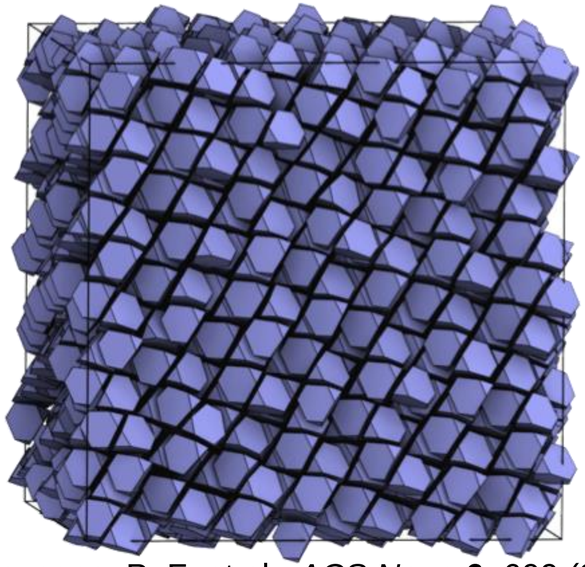
Dihedral/Improper forces
- Harmonic
- Table

Hardware support
- All recent NVIDIA GPUs
- Multi-GPU with MPI
- Multi-CPU with MPI

HPMC - Massively parallel MC on the GPU

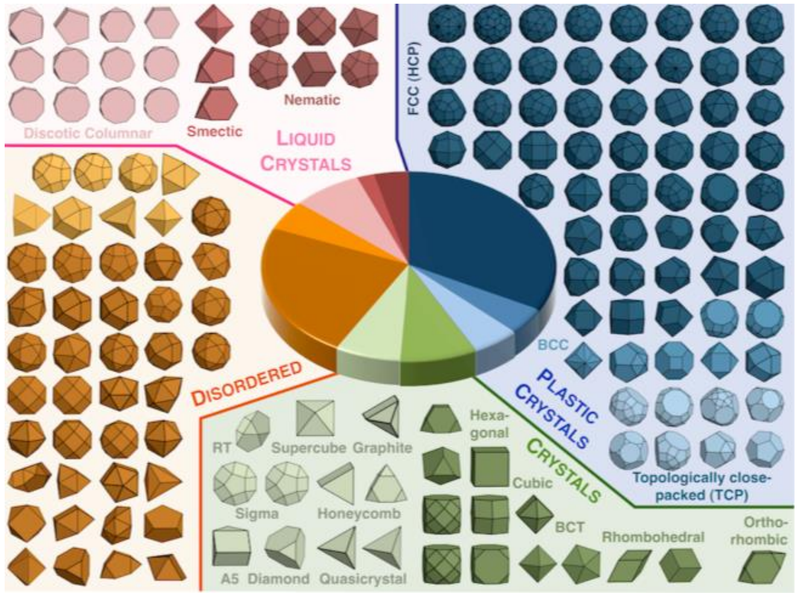


Damasceno et al., *Science* (2012)

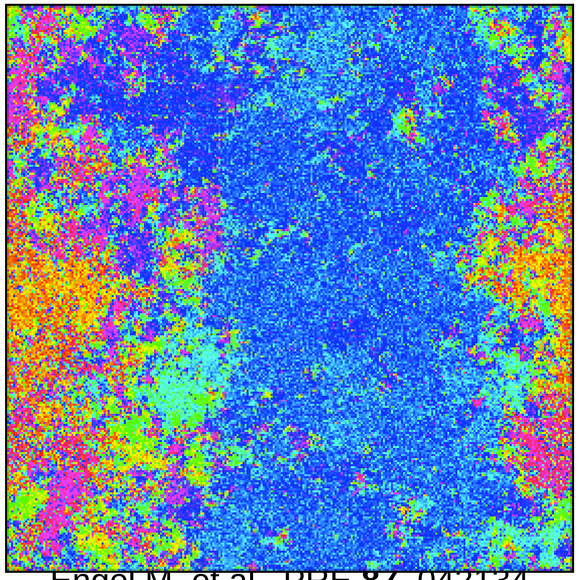


Damasceno, P. F. et al., *ACS Nano* **6**, 609 (2012)

- Hard Particle Monte Carlo plugin for HOOMD-blue
- 2D Shapes
  - Disk
  - Convex (Sphero)polygon
  - Concave polygon
  - Ellipse
- 3D Shapes
  - Sphere
  - Ellipsoid
  - Convex (Sphero)polyhedon
- NVT and NPT ensembles
- Frenkel-Ladd free energy
- Parallel execution on a single GPU
- Domain decomposition across multiple nodes (CPUs or GPUs)



Damasceno et al., *Science* (2012)



Engel M. et al., PRE **87**, 042134 (2013)

Example job script

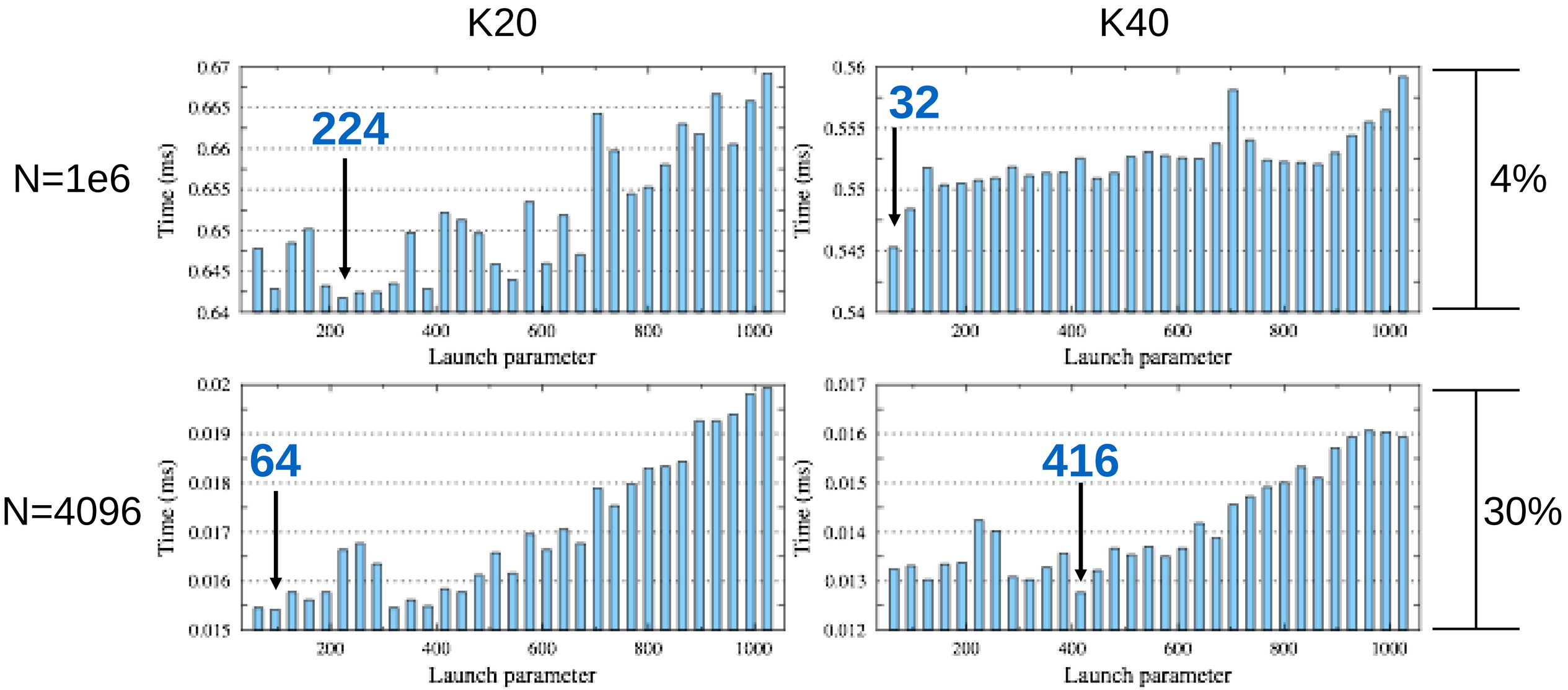
```
from hoomd_script import *
from hoomd_plugins import hpmc

init.read_xml(filename='init.xml')

mc = hpmc.integrate.convex_polygon(seed=10, d=0.25, a=0.3);
mc.shape_param.set('A', vertices=[(-0.5, -0.5), (0.5, -0.5),
                                  (0.5, 0.5), (-0.5, 0.5)]);

run(10e3)
```

Kernel performance depends on launch parameters



Kernel performance depends on launch parameters

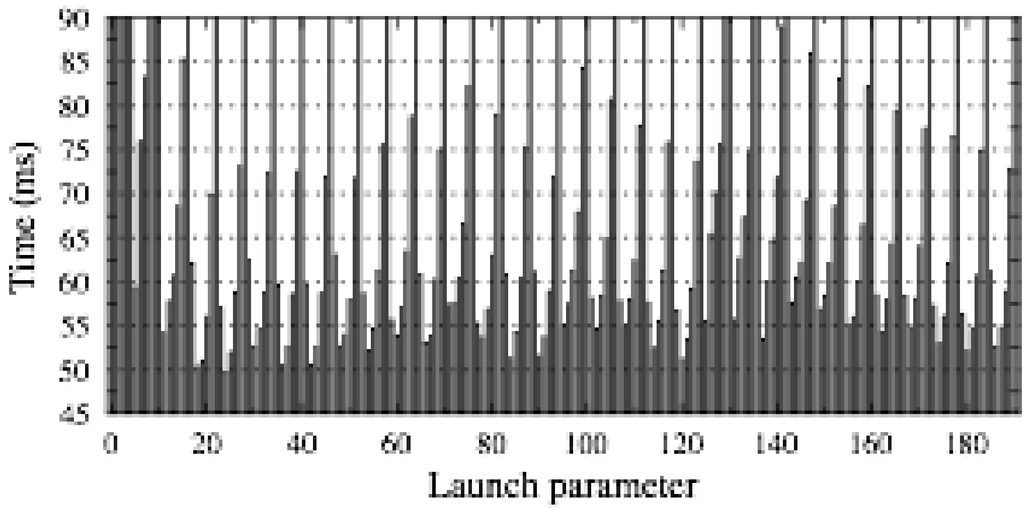
160,2

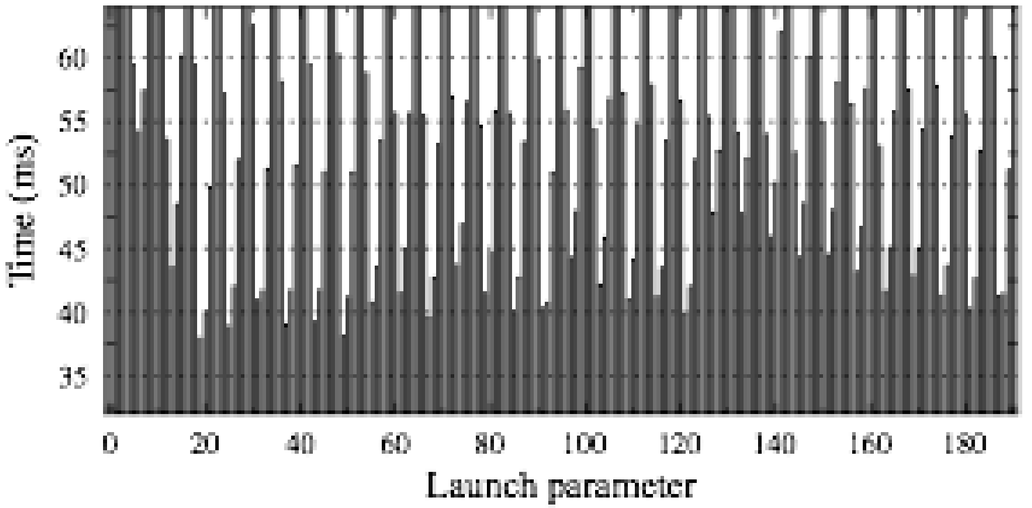
K20

K40

128,4

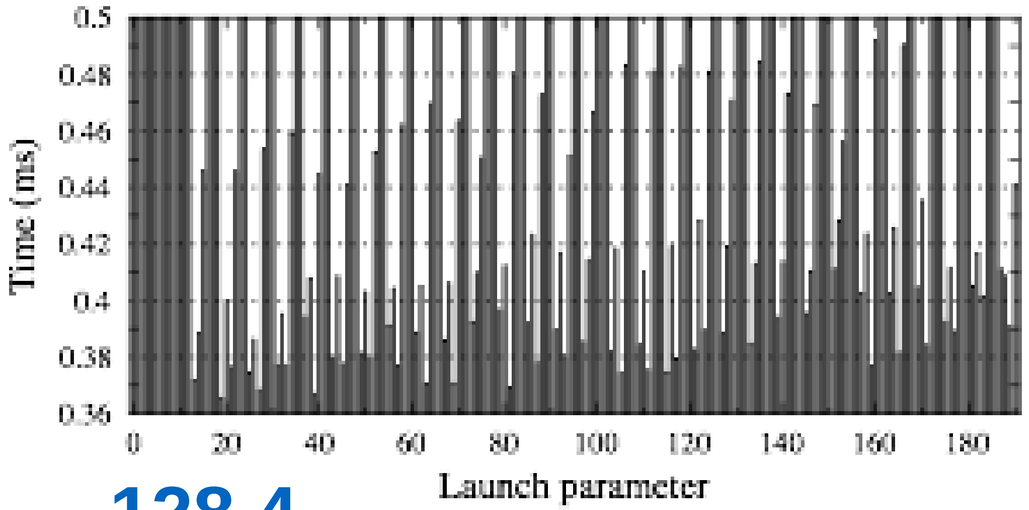
N=1e6

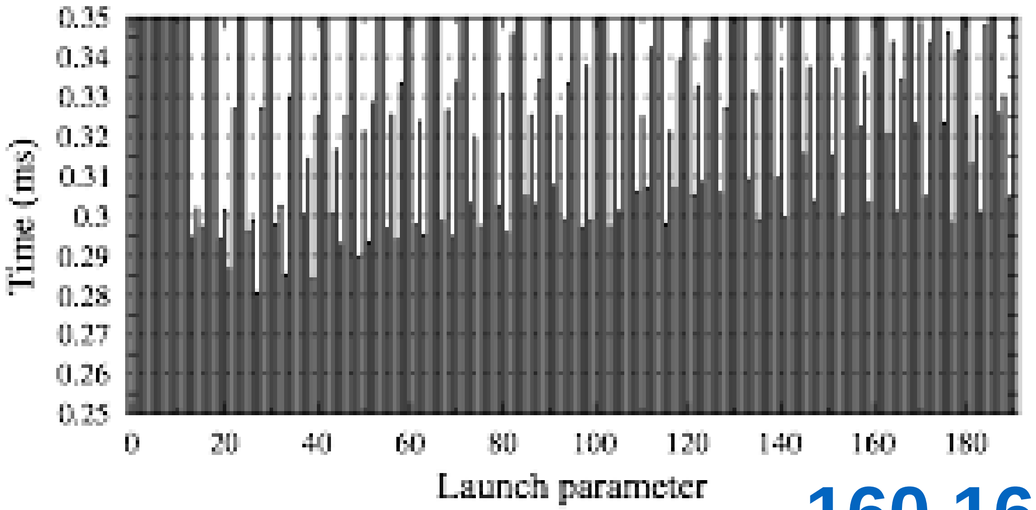




200%

N=4096





40%

128,4

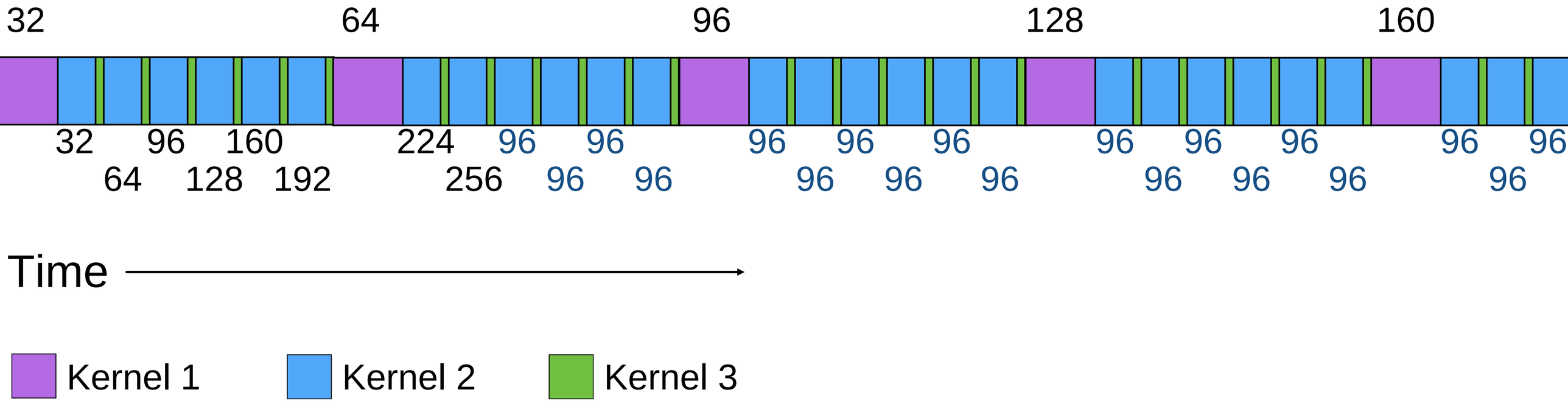
160,16

The need for runtime autotuning

- 100+ kernels, many with multiple variants
- Many GPU generations
- Variations within generations (K20, K40, K80)
- Different CUDA compiler versions (5.5, 6.0, 6.5, 7.0, …)
- Infinite workloads based on user configuration
- Workloads can vary during a single run
- Multiple dimensions of launch parameters (block size, stride, alternate algorithms, …)
- Performance vs launch parameter is not predictable

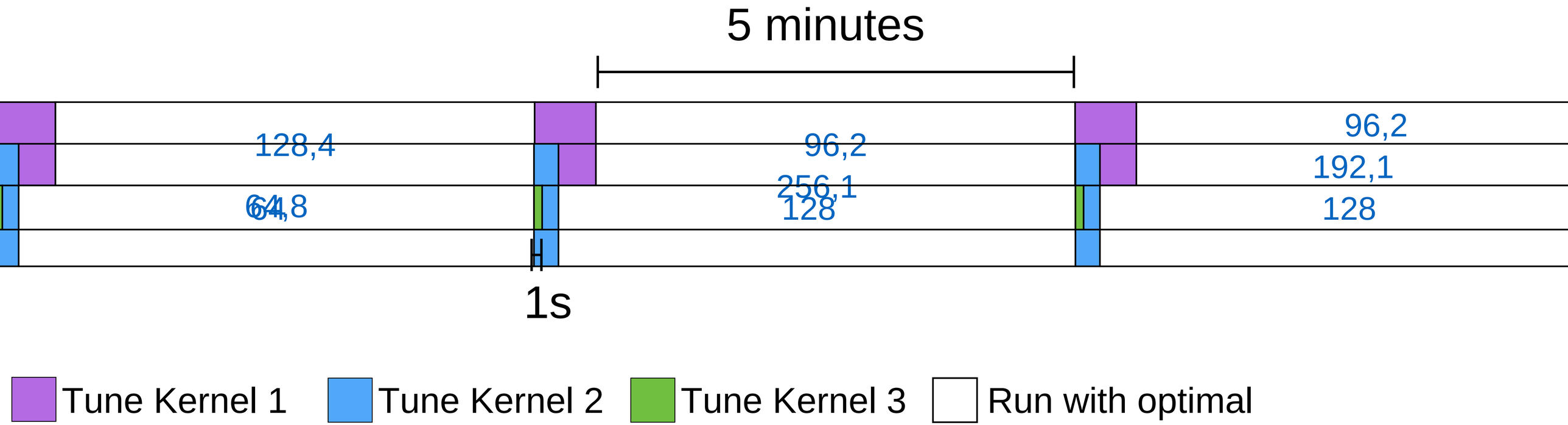
Time steps

- Tuning occurs during actual simulation run steps
- Each kernel has a separate Autotuner
- Kernels may be called at different rates



Repeated tuning

- One scan through launch parameters takes ~1 second
- Lock to the optimal for 5 minutes (user configurable)
- Sample again to adapt to changes
- Run at non-optimal sizes for less than 0.2% of the run



Autotuner interface

```
constructor()
    m_tuner = Autotuner(valid_launch_params)

update(timestep)
    m_tuner.begin()
    call_kernel(…, m_tuner.getParam())
    m_tuner.end()
```

- Minimal additional code to a module
- Initialize tuner
- Wrap the kernel around calls to begin() and end()

Implementation details

- Autotuner methods operate a state machine to control
- When not tuning:
  - getParam() returns the found optimal params
  - begin() and end() are no-ops
- When tuning:
  - getParam() switches to a new parameter on each call
  - begin() and end() use CUDA events to measure time

Code

```
void Autotuner::begin()
    {
    if (m_state == STARTUP || m_state == SCANNING)
       cudaEventRecord(m_start, 0);
   }

void Autotuner::end()
    {
    if (m_state == STARTUP || m_state == SCANNING)
        {
        cudaEventRecord(m_stop, 0);
        cudaEventSynchronize(m_stop);
        cudaEventElapsedTime(&m_samples[m_current_element][m_current_sample],
m_start, m_stop);
        }
    // ... implement state machine update
    }
```

Sampling

- Noise in kernel launch time
- Record M samples per launch param (i.e. 5)
- Take median, mean, or max

- Warmup phase needs to sample len(valid_launch_params)*M launches
- Subsequent runs only need to replace one of the sets of samples or len(valid_launch_params) samples.
- Typically only 32-192

Invalid block sizes

- What about invalid block sizes?
- Not all kernels can be run at every possible block size
- A simple approach
  - Put all possible params in valid_launch_params
  - Clamp to the max possible block size
  - Account for dynamic shared memory if used

```
cudaFuncAttributes attr;
cudaFuncGetAttributes(&attr, kernel<template params>);
int block_size = min(attr.maxThreadsPerBlock, target_block_size);
```

Drawbacks

- Slow period kernels can take minutes to fully tune
- *Runtime* auto-tuning only works with iterative methods
  - Other codes can tune offline
- Floating point reduction kernels give non-deterministic results

- Code available:
  - Autotuner.h / Autotuner.cc in HOOMD-blue
  - http://codeblue.umich.edu/hoomd-blue

Funding / Resources

- Research supported by the National Science Foundation, Division of Materials Research Award # DMR 1409620.

email: joaander@umich.edu