DM-HEOM:
A Portable and Scalable Solver-Framework for the Hierarchical Equations of Motion

Matthias Noack (noack@zib.de), Alexander Reinefeld, Tobias Kramer, Thomas Steinke

Zuse Institute Berlin
Distributed Algorithms and Supercomputing
Motivation

Observations:
- many HPC applications are *legacy code*
  - written and incrementally improved
  - by *domain scientists*
  - often without (modern) software engineering
Motivation

Observations:

- many HPC applications are **legacy code**
  - written and incrementally improved **by domain scientists**
  - often without (modern) software engineering

- hardware life time: **5 years** . . . increasing variety

- software life time: **multiple tens of years** ⇒ outlives systems by far

⇒ need for **code modernisation**
Motivation

Observations:
- many HPC applications are legacy code
  ⇒ written and incrementally improved by domain scientists
  ⇒ often without (modern) software engineering
- hardware life time: 5 years . . . increasing variety
- software life time: multiple tens of years ⇒ outlives systems by far
  ⇒ need for code modernisation

What can we do better?
Motivation

Observations:
- many HPC applications are **legacy code**
  ⇒ written and incrementally improved by domain scientists
  ⇒ often without (modern) software engineering
- hardware life time: 5 years ... increasing variety
- software life time: **multiple tens of years** ⇒ outlives systems by far
  ⇒ need for **code modernisation**

What can we do better? – Our Suggestion:
- **portability first**, then portable performance
Motivation

Observations:
- many HPC applications are legacy code
  - written and incrementally improved by domain scientists
  - often without (modern) software engineering
- hardware life time: 5 years . . . increasing variety
- software life time: multiple tens of years ⇒ outlives systems by far
  ⇒ need for code modernisation

What can we do better? – Our Suggestion:
- portability first, then portable performance
- interdisciplinary collaborations where computer scientists design the software
Motivation

Observations:
- many HPC applications are **legacy code**
  ⇒ written and incrementally improved
    by domain scientists
  ⇒ often without (modern) software engineering
- hardware life time: 5 years . . . increasing variety
- software life time: **multiple tens of years** ⇒ outlives systems by far
  ⇒ need for **code modernisation**

What can we do better? – Our Suggestion:
- **portability first**, then portable performance
- **interdisciplinary collaborations** where computer scientists design the software
- **modern methods and technologies** with a community beyond HPC
Contributions

a) 1st Distributed Memory implementation of the HEOM method: DM-HEOM
Contributions

a) 1st Distributed Memory implementation of the HEOM method: DM-HEOM

b) Interdisciplinary development workflow
Contributions

a) 1st Distributed Memory implementation of the HEOM method: DM-HEOM

b) Interdisciplinary development workflow

c) Design guidelines/experiences for performance portable HPC applications
HEOM - Hierarchical Equations of Motion

Model for Open Quantum Systems

- understand the energy transfer in photo-active molecular complexes
  ⇒ e.g. **photosynthesis**

...but also **quantum computing**
HEOM - Hierarchical Equations of Motion

Model for Open Quantum Systems

- understand the energy transfer in photo-active molecular complexes ⇒ e.g. photosynthesis
  
  ...but also quantum computing

- millions of coupled ODEs

\[
\frac{d\sigma_u}{dt} = -\frac{i}{\hbar} [H, \sigma_u] \\
- \sigma_u \sum_{b=1}^{B} \sum_{k}^{K-1} n_{u,(b,k)} \gamma(b,k) \\
- \sum_{b=1}^{B} \left[ \frac{2\lambda_b}{\beta \hbar^2 \nu_b} - \sum_{k}^{K-1} \frac{c(b,k)}{\hbar \gamma(b,k)} \right] V_{s(b)} V_{s(b)}^\times \sigma_u \\
+ \sum_{b=1}^{B} \sum_{k}^{K-1} i V_{s(b)}^\times \sigma_{u,b,k}^+ \\
+ \sum_{b=1}^{B} \sum_{k}^{K-1} n_{u,(b,k)} \theta_{MA(b,k)} \sigma_{(u,b,k)}^- 
\]
HEOM - Hierarchical Equations of Motion

Model for Open Quantum Systems

- understand the energy transfer in photo-active molecular complexes
  ⇒ e.g. photosynthesis
  ...but also quantum computing
- millions of coupled ODEs

\[
\frac{d\sigma_u}{dt} = -\frac{i}{\hbar} [H, \sigma_u] \quad \text{(LvN commutator)}
+ \sum_{\text{baths}} A\sigma_u \quad \text{(same node)}
+ \sum_{\text{baths}} B\sigma_{u+} \quad \text{(links to layer+1)}
+ \sum_{\text{baths}} C\sigma_{u-} \quad \text{(links to layer-1)}
\]
HEOM - Hierarchical Equations of Motion

Model for Open Quantum Systems

- understand the energy transfer in photo-active molecular complexes
  ⇒ e.g. photosynthesis
  ...but also quantum computing

- millions of coupled ODEs

- hierarchical graph of complex matrices
  (auxiliary density operators, ADOs)
  ⇒ dim: \( N_{\text{sites}} \times N_{\text{sites}} \)
  ⇒ count: exp. in hierarchy depth \( d \)

\[
\frac{d\sigma_u}{dt} = -\frac{i}{\hbar} [H, \sigma_u] \quad \text{(LvN commutator)}
+ \sum_{\text{baths}} A\sigma_u \quad \text{(same node)}
+ \sum_{\text{baths}} B\sigma_u^+ \quad \text{(links to layer+1)}
+ \sum_{\text{baths}} C\sigma_u^- \quad \text{(links to layer-1)}
\]
HEOM - Computational Properties

- **ODE**: dominated by commutator term: 
  \[ [H, \sigma_u] = H\sigma_u - \sigma_u H \]
HEOM - Computational Properties

- **ODE**: dominated by commutator term: 
  \[ [H, \sigma_u] = H\sigma_u - \sigma_u H \]
  - for each \( \sigma_u \): \( 16N_s^3 \) FLOP per \( 2 \cdot 2 \cdot 8 \cdot N_s^2 \) Byte

  \( \Rightarrow \) arithmetic intensity: \( \frac{N_s}{2} \text{ FLOP} \text{ Byte} \) \( \Rightarrow \) compute bound for larger \( N_s \) sites

<table>
<thead>
<tr>
<th>Device Name (architecture)</th>
<th>[TFLOPS]</th>
<th>[GiB/s]</th>
<th>[FLOP/Byte]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel Xeon Gold 6138 (SKL)</td>
<td>2.56</td>
<td>238</td>
<td>10.8</td>
</tr>
<tr>
<td>Intel Xeon E5-2680v3 (HSW)</td>
<td>0.96</td>
<td>136</td>
<td>7.1</td>
</tr>
<tr>
<td>Intel Xeon Phi 7250 (KNL)</td>
<td>2.61</td>
<td>490/115</td>
<td>5.3/22.7</td>
</tr>
<tr>
<td>Nvidia Tesla K40 (Kepler)</td>
<td>1.31</td>
<td>480</td>
<td>2.7</td>
</tr>
<tr>
<td>AMD Firepro W8100 (Hawaii)</td>
<td>2.1</td>
<td>320</td>
<td>6.6</td>
</tr>
</tbody>
</table>

\( a \) Assuming 1.2 GHz AVX frequency.  
\( b \) On-chip MCDRAM / DRAM
HEOM - Computational Properties

- **ODE**: dominated by commutator term: $[H, \sigma_u] = H\sigma_u - \sigma_u H$
  - for each $\sigma_u$: $16N_{\text{sites}}^3$ FLOP per $2 \cdot 2 \cdot 8 \cdot N_{\text{sites}}^2$ Byte

  $\Rightarrow$ arithmetic intensity: $\frac{N_{\text{sites}}}{2}$ FLOP/Byte $\Rightarrow$ compute bound for larger $N_{\text{sites}}$

- **numerical integration** (weighted add):

<table>
<thead>
<tr>
<th>Device Name (architecture)</th>
<th>TFLOPS</th>
<th>GiB/s</th>
<th>FLOP/Byte</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 $\times$ Intel Xeon Gold 6138 (SKL)</td>
<td>2.56</td>
<td>238</td>
<td>10.8</td>
</tr>
<tr>
<td>2 $\times$ Intel Xeon E5-2680v3 (HSW)</td>
<td>0.96</td>
<td>136</td>
<td>7.1</td>
</tr>
<tr>
<td>Intel Xeon Phi 7250 (KNL)</td>
<td>2.61</td>
<td>1490/115</td>
<td>5.3/22.7</td>
</tr>
<tr>
<td>Nvidia Tesla K40 (Kepler)</td>
<td>1.31</td>
<td>480</td>
<td>2.7</td>
</tr>
<tr>
<td>AMD Firepro W8100 (Hawaii)</td>
<td>2.1</td>
<td>320</td>
<td>6.6</td>
</tr>
</tbody>
</table>

*a* Assuming 1.2 GHz AVX frequency.

*b* On-chip MCDRAM / DRAM
HEOM - Computational Properties

- **ODE**: dominated by commutator term: $[H, \sigma_u] = H\sigma_u - \sigma_u H$
  - for each $\sigma_u$: $16N_{\text{sites}}^3$ FLOP per $2 \cdot 2 \cdot 8 \cdot N_{\text{sites}}^2$ Byte
  
  $\Rightarrow$ arithmetic intensity: $\frac{N_{\text{sites}}}{2}$ FLOP/Byte $\Rightarrow$ compute bound for larger $N_{\text{sites}}$

- **numerical integration** (weighted add):
  
  $\Rightarrow$ arithmetic intensity: $\frac{1}{8}$ FLOP/Byte $\Rightarrow$ memory bound on relevant hardware
HEOM - Computational Properties

- **ODE**: dominated by commutator term: 
  \[ [H, \sigma_u] = H\sigma_u - \sigma_u H \]
  - for each \( \sigma_u \): \( 16N_{\text{sites}}^3 \) FLOP per \( 2 \cdot 2 \cdot 8 \cdot N_{\text{sites}}^2 \) Byte
  
  \( \Rightarrow \) arithmetic intensity: \( \frac{N_{\text{sites}}}{2} \cdot \frac{\text{FLOP}}{\text{Byte}} \) \( \Rightarrow \) compute bound for larger \( N_{\text{sites}} \)

- **Numerical integration** (weighted add):
  
  \( \Rightarrow \) arithmetic intensity: \( \frac{1}{8} \cdot \frac{\text{FLOP}}{\text{Byte}} \) \( \Rightarrow \) memory bound on relevant hardware

<table>
<thead>
<tr>
<th>Device Name (architecture)</th>
<th>compute [TFLOPS]</th>
<th>memory bw. [GiB/s]</th>
<th>[FLOP/Byte]</th>
</tr>
</thead>
<tbody>
<tr>
<td>2( \times ) Intel Xeon Gold 6138 (SKL)</td>
<td>2.56</td>
<td>238</td>
<td>10.8</td>
</tr>
<tr>
<td>2( \times ) Intel Xeon E5-2680v3 (HSW)</td>
<td>0.96</td>
<td>136</td>
<td>7.1</td>
</tr>
<tr>
<td>Intel Xeon Phi 7250 (KNL)</td>
<td>2.61(^a)</td>
<td>490/115(^b)</td>
<td>5.3/22.7(^b)</td>
</tr>
<tr>
<td>Nvidia Tesla K40 (Kepler)</td>
<td>1.31</td>
<td>480</td>
<td>2.7</td>
</tr>
<tr>
<td>AMD Firepro W8100 (Hawaii)</td>
<td>2.1</td>
<td>320</td>
<td>6.6</td>
</tr>
</tbody>
</table>

\(^a\)Assuming 1.2 GHz AVX frequency. \(^b\)On-chip MCDRAM / DRAM
## HEOM - Example Systems

<table>
<thead>
<tr>
<th>system name</th>
<th>$N_{sites}$</th>
<th>baths per site</th>
<th>depth $d$</th>
<th>total ADO memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>FMO</td>
<td>7</td>
<td>22</td>
<td>8</td>
<td>2.8 GiB</td>
</tr>
<tr>
<td>LHC II monomer</td>
<td>14</td>
<td>1</td>
<td>3</td>
<td>5.6 GiB</td>
</tr>
<tr>
<td>PS I</td>
<td>96</td>
<td>1</td>
<td>3</td>
<td>129.2 GiB</td>
</tr>
<tr>
<td>PS I</td>
<td>96</td>
<td>1</td>
<td>4</td>
<td>3231.0 GiB</td>
</tr>
</tbody>
</table>

memory consumption assuming an RK4 solver

⇒ larger systems cannot be solved on a single node:
• memory footprint
• time to solution
⇒ distributed memory implementation required
<table>
<thead>
<tr>
<th>system name</th>
<th>$N_{\text{sites}}$</th>
<th>baths per site</th>
<th>depth $d$</th>
<th>total ADO memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>FMO</td>
<td>7</td>
<td>22</td>
<td>8</td>
<td>2.8 GiB</td>
</tr>
<tr>
<td>LHC II monomer</td>
<td>14</td>
<td>1</td>
<td>3</td>
<td>5.6 GiB</td>
</tr>
<tr>
<td>PS I</td>
<td>96</td>
<td>1</td>
<td>3</td>
<td>129.2 GiB</td>
</tr>
<tr>
<td>PS I</td>
<td>96</td>
<td>1</td>
<td>4</td>
<td>3231.0 GiB</td>
</tr>
</tbody>
</table>

memory consumption assuming an RK4 solver

⇒ larger systems **cannot** be solved on a single node:
  - memory footprint
  - time to solution
### HEOM - Example Systems

<table>
<thead>
<tr>
<th>system name</th>
<th>$N_{\text{sites}}$</th>
<th>baths per site</th>
<th>depth $d$</th>
<th>total ADO memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>FMO</td>
<td>7</td>
<td>22</td>
<td>8</td>
<td>2.8 GiB</td>
</tr>
<tr>
<td>LHC II monomer</td>
<td>14</td>
<td>1</td>
<td>3</td>
<td>5.6 GiB</td>
</tr>
<tr>
<td>PS I</td>
<td>96</td>
<td>1</td>
<td>3</td>
<td>129.2 GiB</td>
</tr>
<tr>
<td>PS I</td>
<td>96</td>
<td>1</td>
<td>4</td>
<td>3231.0 GiB</td>
</tr>
</tbody>
</table>

Memory consumption assuming an RK4 solver:

- Larger systems **cannot** be solved on a single node:
  - Memory footprint
  - Time to solution

⇒ Distributed memory implementation **required**
Interdisciplinary Workflow

Mathematical Model

domain experts
computer scientists
Interdisciplinary Workflow

Mathematical Model

- ODEs
- PDEs
- Graphs
- ...
Interdisciplinary Workflow

Mathematical Model
- ODEs
- PDEs
- Graphs
- ...

High-Level Prototype (Mathematica)
Interdisciplinary Workflow

Mathematical Model
- ODEs
- PDEs
- Graphs
- ...

High-Level Prototype (Mathematica)
- domain scientist’s tool
- high level
- symbolic solvers
- arbitrary precision
- very limited performance
Interdisciplinary Workflow

Mathematical Model
- ODEs
- PDEs
- Graphs
- ...

High-Level Prototype (Mathematica)
- domain scientist’s tool
- high level
- symbolic solvers
- arbitrary precision
- very limited performance

OpenCL kernel within Mathematica
Interdisciplinary Workflow

- ODEs
- PDEs
- Graphs
- ...

- domain scientist’s tool
- high level
- symbolic solvers
- arbitrary precision
- very limited performance

- replace some code with OpenCL
OpenCL (Open Computing Language) in a Nutshell

- open, royalty-free **standard** for cross-platform, **parallel programming**
- maintained by **Khronos Group**

- personal computers, servers, mobile devices and embedded platforms
- first released: 2009-08-28

https://www.khronos.org/opencl/
OpenCL (Open Computing Language) in a Nutshell

- open, royalty-free **standard** for cross-platform, **parallel programming**
- maintained by **Khronos Group**
- personal computers, servers, mobile devices and embedded platforms
- first released: 2009-08-28

[Visit the Khronos Group website for more information](https://www.khronos.org/opencl/)
OpenCL Platform and Memory Model

Compute Device (CD)  Compute Unit (CU)

Processing Element (PE)

Memory Model:
• CD has device memory with glottal/constant addr. space
• CU has local memory addr. space
• PE has private memory addr. space
⇒ relaxed consistency
**OpenCL Platform and Memory Model**

**Memory Model:**
- CD has device memory with *glottal/constant* addr. space
- CU has *local* memory addr. space
- PE has *private* memory addr. space
  ⇒ relaxed consistency
## OpenCL Machine Model Mapping

<table>
<thead>
<tr>
<th>OpenCL Platform</th>
<th>CPU Hardware</th>
<th>GPU Hardware</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Compute Device</strong></td>
<td>Processor/Board</td>
<td>GPU device</td>
</tr>
<tr>
<td><strong>Compute Unit</strong></td>
<td>Core (thread)</td>
<td>Streaming MP</td>
</tr>
<tr>
<td><strong>Processing Element</strong></td>
<td>SIMD Lane</td>
<td>CUDA Core</td>
</tr>
<tr>
<td>global/const. memory</td>
<td>DRAM</td>
<td>DRAM</td>
</tr>
<tr>
<td>local memory</td>
<td>DRAM</td>
<td>Shared Memory</td>
</tr>
<tr>
<td>private memory</td>
<td>Register/DRAM</td>
<td>Priv. Mem./Register</td>
</tr>
</tbody>
</table>
### OpenCL Machine Model Mapping

<table>
<thead>
<tr>
<th>OpenCL Platform</th>
<th>CPU Hardware</th>
<th>GPU Hardware</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Compute Device</strong></td>
<td>Processor/Board</td>
<td>GPU device</td>
</tr>
<tr>
<td><strong>Compute Unit</strong></td>
<td>Core (thread)</td>
<td>Streaming MP</td>
</tr>
<tr>
<td><strong>Processing Element</strong></td>
<td>SIMD Lane</td>
<td>CUDA Core</td>
</tr>
<tr>
<td><strong>global/const. memory</strong></td>
<td>DRAM</td>
<td>DRAM</td>
</tr>
<tr>
<td><strong>local memory</strong></td>
<td>DRAM</td>
<td>Shared Memory</td>
</tr>
<tr>
<td><strong>private memory</strong></td>
<td>Register/DRAM</td>
<td>Priv. Mem./Register</td>
</tr>
</tbody>
</table>

⇒ write code for this **abstract machine model**
**OpenCL Machine Model Mapping**

<table>
<thead>
<tr>
<th>OpenCL Platform</th>
<th>CPU Hardware</th>
<th>GPU Hardware</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Compute Device</strong></td>
<td>Processor/Board</td>
<td>GPU device</td>
</tr>
<tr>
<td><strong>Compute Unit</strong></td>
<td>Core (thread)</td>
<td>Streaming MP</td>
</tr>
<tr>
<td><strong>Processing Element</strong></td>
<td>SIMD Lane</td>
<td>CUDA Core</td>
</tr>
<tr>
<td>global/const. memory</td>
<td>DRAM</td>
<td>DRAM</td>
</tr>
<tr>
<td>local memory</td>
<td>DRAM</td>
<td>Shared Memory</td>
</tr>
<tr>
<td>private memory</td>
<td>Register/DRAM</td>
<td>Priv. Mem./Register</td>
</tr>
</tbody>
</table>

⇒ write code for this **abstract machine model**
⇒ device-specific OpenCL compiler and runtime maps it to **actual hardware**
### OpenCL Machine Model Mapping

<table>
<thead>
<tr>
<th>OpenCL Platform</th>
<th>CPU Hardware</th>
<th>GPU Hardware</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compute Device</td>
<td>Processor/Board</td>
<td>GPU device</td>
</tr>
<tr>
<td>Compute Unit</td>
<td>Core (thread)</td>
<td>Streaming MP</td>
</tr>
<tr>
<td>Processing Element</td>
<td>SIMD Lane</td>
<td>CUDA Core</td>
</tr>
<tr>
<td>global/const. memory</td>
<td>DRAM</td>
<td>DRAM</td>
</tr>
<tr>
<td>local memory</td>
<td>DRAM</td>
<td>Shared Memory</td>
</tr>
<tr>
<td>private memory</td>
<td>Register/DRAM</td>
<td>Priv. Mem./Register</td>
</tr>
</tbody>
</table>

⇒ write code for this **abstract machine model**
⇒ device-specific OpenCL compiler and runtime maps it to **actual hardware**
⇒ **library-only** implementation: no toolchain, many language bindings
OpenCL Machine Model Mapping

<table>
<thead>
<tr>
<th>OpenCL Platform</th>
<th>CPU Hardware</th>
<th>GPU Hardware</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compute Device</td>
<td>Processor/Board</td>
<td>GPU device</td>
</tr>
<tr>
<td>Compute Unit</td>
<td>Core (thread)</td>
<td>Streaming MP</td>
</tr>
<tr>
<td>Processing Element</td>
<td>SIMD Lane</td>
<td>CUDA Core</td>
</tr>
<tr>
<td>global/const. memory</td>
<td>DRAM</td>
<td>DRAM</td>
</tr>
<tr>
<td>local memory</td>
<td>DRAM</td>
<td>Shared Memory</td>
</tr>
<tr>
<td>private memory</td>
<td>Register/DRAM</td>
<td>Priv. Mem./Register</td>
</tr>
</tbody>
</table>

⇒ write code for this abstract machine model
⇒ device-specific OpenCL compiler and runtime maps it to actual hardware
⇒ library-only implementation: no toolchain, many language bindings
⇒ currently: widest practical portability of parallel programming models
Interdisciplinary Workflow

Mathematical Model
- ODEs
- PDEs
- Graphs
- ...

High-Level Prototype (Mathematica)
- domain scientist’s tool
- high level
- symbolic solvers
- arbitrary precision
- very limited performance

OpenCL kernel within Mathematica
- replace some code with OpenCL
- compare results
- figure out numerics
- use accelerators in MM
Interdisciplinary Workflow

Mathematical Model
- ODEs
- PDEs
- Graphs
- ...

High-Level Prototype (Mathematica)
- domain scientist’s tool
- high level
- symbolic solvers
- arbitrary precision
- very limited performance

OpenCL kernel within Mathematica
- replace some code with OpenCL
- compare results
- figure out numerics
- use accelerators in MM

C++ App. with OpenCL kernel

…
DM-HEOM: Design Goals and Principles

- **portability**
  - modern, *vendor-independent* standards: C++11/14/17, OpenCL, MPI-3, CMake, ...
DM-HEOM: Design Goals and Principles

- **portability**
  - modern, *vendor-independent* standards: C++11/14/17, OpenCL, MPI-3, CMake, ...

- **performance portability**
  - flexibility: make performance-critical aspects configurable
  - avoid device-specific code branches (i.e. redundant code)
DM-HEOM: Design Goals and Principles

- **portability**
  - modern, *vendor-independent* standards: C++11/14/17, OpenCL, MPI-3, CMake, ...

- **performance portability**
  - flexibility: make performance-critical aspects configurable
  - avoid device-specific code branches (i.e. redundant code)

- **scalability**
  - from single nodes to supercomputers
  - a single code base for small and large problems
DM-HEOM: Design Goals and Principles

- **portability**
  - modern, *vendor-independent* standards: C++11/14/17, OpenCL, MPI-3, CMake, ...

- **performance portability**
  - flexibility: make performance-critical aspects configurable
  - avoid device-specific code branches (i.e. redundant code)

- **scalability**
  - from single nodes to supercomputers
  - a single code base for small and large problems

- powerful high-level **abstractions**
  - small applications on top
  - easily understood by domain scientists
DM-HEOM: Design Goals and Principles

- **portability**
  - modern, *vendor-independent* standards: C++11/14/17, OpenCL, MPI-3, CMake, ...

- **performance portability**
  - flexibility: make performance-critical aspects configurable
  - avoid device-specific code branches (i.e. redundant code)

- **scalability**
  - from single nodes to supercomputers
  - a single code base for small and large problems

- powerful high-level **abstractions**
  - small applications on top
  - easily understood by domain scientists

- **separation and exchangeability** of different aspects and strategies
  - partitioning, numerical methods, memory layout, parallelisation, communication, etc.
Interdisciplinary Workflow

Mathematical Model
- ODEs
- PDEs
- Graphs
- ...

High-Level Prototype (Mathematica)
- domain scientist’s tool
- high level
- symbolic solvers
- arbitrary precision
- very limited performance

OpenCL kernel within Mathematica
- replace some code with OpenCL
- compare results
- figure out numerics
- use accelerators in MM

C++ App. with OpenCL kernel
Interdisciplinary Workflow

- ODEs
- PDEs
- Graphs
- ...

High-Level Prototype (Mathematica)
- domain scientist’s tool
- high level
- symbolic solvers
- arbitrary precision
- very limited performance

OpenCL kernel within Mathematica
- replace some code with OpenCL
- compare results
- figure out numerics
- use accelerators in MM

C++ App. with OpenCL kernel
- start single node
- OpenCL 1.2 for hotspots
- modern C++ 11/14/17
- CMake for building
HEOM Config describes physics
Hierarchy Graph is used to initialise a problem

\[ \text{ODE} \] (HEOM formula) \[ \Rightarrow \] encapsulates OpenCL kernel code

OpenCL Config specifies runtime configuration

Solver works on ODE \[ \Rightarrow \] encapsulates OpenCL runtime \[ \Rightarrow \] encapsulates numerics \[ \Rightarrow \] produces Results
Distributed Memory HEOM: Single Node C++ Application

- **HEOM Config** describes physics
- **Hierarchy Graph** is used to initialise a problem
- **ODE (HEOM formula)** encapsulates OpenCL kernel code
- **OpenCL Config** specifies runtime configuration
- **Solver** works on **ODE** encapsulates OpenCL runtime encapsulates numerics produces **Results**
Distributed Memory HEOM: Single Node C++ Application

- **HEOM Config** describes physics
- **Hierarchy Graph** is used to initialise a problem **Instance**
Distributed Memory HEOM: Single Node C++ Application

- **HEOM Config** describes physics
- **Hierarchy Graph** is used to initialise a problem **Instance**
- **ODE (HEOM formula)**
  - $\Rightarrow$ encapsulates OpenCL kernel code

**Diagram:**

1. HEOM Config
2. HEOM Configuration
3. Hierarchy Graph
4. Instance
5. Solver
6. ODE
7. Results

- HEOM Config
- Hierarchy Graph
- Instance
- ODE
- Results
Distributed Memory HEOM: Single Node C++ Application

- **HEOM Config** describes physics
- **Hierarchy Graph** is used to initialise a problem **Instance**
- **ODE** (HEOM formula) ⇒ encapsulates OpenCL kernel code
- **OpenCL Config** specifies runtime configuration

- **HEOM Configuration**
- **Hierarchy Graph**
- **Instance**
- **Solver**, **ODE**
- **Results**
- **HEOM Config** describes physics
- **Hierarchy Graph** is used to initialise a problem *Instance*
- **ODE** (HEOM formula) 
  \[ \Rightarrow \] encapsulates OpenCL kernel code
- **OpenCL Config** specifies runtime configuration
- **Solver** works on **ODE** 
  \[ \Rightarrow \] encapsulates OpenCL runtime 
  \[ \Rightarrow \] encapsulates numerics 
  \[ \Rightarrow \] produces *Results*
From Portability to Performance

OpenCL:

- guarantees portability...
- ... but no portable performance
From Portability to Performance

OpenCL:
- guarantees portability . . .
- . . . but no portable performance

Strategy:
1. identify key optimisations each device requires
2. make the code configurable to the device’s needs
   . . . without writing a version for each device
Performance Portability: Node-Level
OpenCL Runtime Kernel Compilation

- necessary for **portability**
Performance Portability: Node-Level

OpenCL Runtime Kernel Compilation

- necessary for **portability**
- exploitable for **performance**
  a) facilitate compiler optimisation
  b) configure code before compilation

Runtime compilation for non-OpenCL codes: [https://github.com/noma/kart](https://github.com/noma/kart)
Performance Portability: Node-Level

OpenCL Runtime Kernel Compilation

- necessary for **portability**
- exploitable for **performance**
  a) facilitate compiler optimisation
  b) configure code before compilation

a) compiler optimisation:

- use host-code runtime-constants as kernel-code compile-time constants
  - e.g. sizes, loop-counts, ...
  ⇒ resolve index computations, eliminate branches, unroll loops, ...

Runtime compilation for non-OpenCL codes: https://github.com/noma/kart
Performance Portability: Node-Level

OpenCL Runtime Kernel Compilation

- necessary for **portability**
- exploitable for **performance**
  a) facilitate compiler optimisation
  b) configure code before compilation

a) compiler optimisation:

- use host-code runtime-constants as kernel-code compile-time constants
  - e.g. sizes, loop-counts, ...
  ⇒ resolve index computations, eliminate branches, unroll loops, ...

b) configurable kernel code:

- work-item granularity
- memory-layout

Runtime compilation for non-OpenCL codes: https://github.com/noma/kart
Performance Portability: Work-item Granularity

- amount of work per OpenCL work-item
- processed on the smallest parallel hardware execution unit (PE)

⇒ most important for efficient device utilisation
Performance Portability: Work-item Granularity

- amount of work per OpenCL work-item
- processed on the smallest parallel hardware execution unit (PE)
  ⇒ most important for efficient device utilisation

GPU Devices

- many SIMT cores
- thousands of light-weight hardware threads
- executed in groups of 32 or 64 thread
  ⇒ one ADO matrix element per thread
Performance Portability: Work-item Granularity

- amount of work per OpenCL work-item
- processed on the smallest parallel hardware execution unit (PE)
  ⇒ most important for efficient device utilisation

GPU Devices

- many SIMT cores
- thousands of light-weight hardware threads
- executed in groups of 32 or 64 thread
  ⇒ one ADO matrix element per thread

CPU Devices

- fewer, more complex general-purpose cores
- SIMD vector units with 4 to 8 lanes
  ⇒ one ADO matrix per processing element
Performance Portability: Work-item Granularity

- amount of work per OpenCL *work-item*
- processed on the smallest parallel hardware execution unit (PE)
  ⇒ most important for efficient device utilisation

**GPU Devices**
- many SIMT cores
- thousands of light-weight hardware threads
- executed in groups of 32 or 64 thread
  ⇒ one ADO matrix element per thread

**CPU Devices**
- fewer, more complex general-purpose cores
- SIMD vector units with 4 to 8 lanes
  ⇒ one ADO matrix per processing element

⇒ requires a *compile-time configurable outer loop-nest* inside the kernel
Performance Portability: Memory Layout

⇒ match requirement of **device-specific memory architecture**
- **GPU**: coalesced access from working-groups without bank-conflicts
- **CPU**: contiguous SIMD vector load/store instructions (avoid gather/scatter ops)
Performance Portability: Memory Layout

⇒ match requirement of **device-specific memory architecture**
  - **GPU**: coalesced access from working-groups without bank-conflicts
  - **CPU**: contiguous SIMD vector load/store instructions (avoid gather/scatter ops)

⇒ **parallelisation strategy**, **granularity** and **memory layout** must match
  - e.g. outer loop vectorisation for SIMD architectures
AoS vs. SIMD-friendly AoSoA Memory Layout

1st matrix: A
A’s 1st row
\[ a_{11}, a_{12}, a_{13}, \ldots, a_{1n}, a_{21}, \ldots, a_{2n} \]
A’s 2nd row
\[ a_{nn}, b_{11}, b_{12}, \ldots, b_{1n}, b_{21}, \ldots, b_{nn} \]

2nd matrix: B
B’s 1st row
\[ b_{11}, b_{12}, \ldots, b_{1n}, b_{21}, \ldots, b_{nn} \]

3rd matrix: C
C’s 1st row
\[ b_{nn}, c_{11}, \ldots, c_{1n}, c_{21}, \ldots, c_{nn} \]

8th matrix: H
8 elements
\[ h_{11}, \ldots, h_{nn} \]

work-item 1 \(\Rightarrow\) mapped to 8 SIMD lanes

costly gather/scatter access
optimal: contiguous load/store
AoS vs. SIMD-friendly AoSoA Memory Layout

1st matrix: A

\[ \begin{array}{ccccccc}
  a_{11} & a_{12} & a_{13} & \cdots & a_{1n} & a_{21} & a_{2n} \\
  a_{nn} & b_{11} & b_{12} & \cdots & b_{1n} & b_{21} & b_{2n} \\
  \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\
  a_{nn} & b_{nn} & c_{11} & \cdots & c_{1n} & c_{21} & c_{nn} \\
  h_{11} & \cdots & h_{nn} & \cdots & \cdots & \cdots & \cdots \\
\end{array} \]

2nd matrix: B

3rd matrix: C

8th matrix: H

work-item\_1 | work-item\_2 | work-item\_3 | \cdots | work-item\_8 \Rightarrow \text{mapped to 8 SIMD lanes}
AoS vs. SIMD-friendly AoSoA Memory Layout

1st matrix: A

A’s 1st row: 
\[ a_{11} \ a_{12} \ a_{13} \ldots a_{1n} \ a_{21} \ldots a_{2n} \ldots \]

A’s 2nd row: 
\[ a_{nn} \ b_{11} \ b_{12} \ldots b_{1n} \ b_{21} \ldots b_{nn} \ldots \]

2nd matrix: B

B’s 1st row: 
\[ b_{nn} \ c_{11} \ldots c_{1n} \ c_{21} \ldots c_{nn} \ldots \]

3rd matrix: C

C’s 1st row: 
\[ c_{nn} \ h_{11} \ldots h_{1n} \ldots h_{nn} \ldots \]

8th matrix: H

work-item_1 \ work-item_2 \ work-item_3 \ldots \ work-item_8

⇒ mapped to 8 SIMD lanes

costly gather/scatter access
**AoS vs. SIMD-friendly AoSoA Memory Layout**

<table>
<thead>
<tr>
<th>1st matrix: A</th>
<th>2nd matrix: B</th>
<th>3rd matrix: C</th>
<th>8th matrix: H</th>
</tr>
</thead>
<tbody>
<tr>
<td>A’s 1st row</td>
<td>B’s 1st row</td>
<td>C’s 1st row</td>
<td>H’s 1st row</td>
</tr>
<tr>
<td>a(<em>{11})a(</em>{12})a(<em>{13}) \ldots a(</em>{1n})a(<em>{21}) \ldots a(</em>{2n})</td>
<td>b(<em>{11})b(</em>{12}) \ldots b(<em>{1n})b(</em>{21}) \ldots</td>
<td>b(<em>{nn})c(</em>{11}) \ldots c(<em>{1n})c(</em>{21}) \ldots</td>
<td>h(<em>{11})h(</em>{nn}) \ldots</td>
</tr>
</tbody>
</table>

- work-item\(_1\) work-item\(_2\) work-item\(_3\) \ldots work-item\(_8\) ⇒ mapped to 8 SIMD lanes

- **costly gather/scatter access**

- 8 elements
- 8 rows
- 8 matrices = 1 SoA ‘package’
AoS vs. SIMD-friendly AoSoA Memory Layout

1st matrix: A

A’s 1st row

\[
\begin{bmatrix}
    a_{11} & a_{12} & a_{13} \\
    a_{1n} & a_{21} & \cdots \\
\end{bmatrix}
\]

A’s 2nd row

\[
\begin{bmatrix}
    a_{2n} \\
    \vdots \\
    a_{nn} \\
\end{bmatrix}
\]

2nd matrix: B

B’s 1st row

\[
\begin{bmatrix}
    a_{nn} & b_{11} & b_{12} \\
    b_{1n} & b_{21} & \cdots \\
\end{bmatrix}
\]

B’s 2nd row

\[
\begin{bmatrix}
    b_{2n} & c_{11} \\
    \cdots & \cdots \\
    c_{nn} \\
\end{bmatrix}
\]

3rd matrix: C

C’s 1st row

\[
\begin{bmatrix}
    b_{nn} & c_{11} & \cdots \\
    c_{1n} & c_{21} & \cdots \\
\end{bmatrix}
\]

C’s 2nd row

\[
\begin{bmatrix}
    c_{2n} \\
    \cdots \\
    c_{nn} \\
\end{bmatrix}
\]

8th matrix: H

H’s 1st row

\[
\begin{bmatrix}
    b_{nn} & c_{11} & \cdots \\
    c_{1n} & c_{21} & \cdots \\
\end{bmatrix}
\]

H’s 2nd row

\[
\begin{bmatrix}
    c_{2n} \\
    \cdots \\
    c_{nn} \\
\end{bmatrix}
\]

\[
\begin{bmatrix}
    h_{11} & h_{nn} \\
    \vdots & \vdots \\
    h_{nn} \\
\end{bmatrix}
\]

work-item\_1 \quad work-item\_2 \quad work-item\_3 \quad \cdots \quad work-item\_8

work-item\_1 \quad work-item\_2 \quad work-item\_3

work-item\_8

\Rightarrow\text{ mapped to 8 SIMD lanes}

costly gather/scatter access

8 elements

8 rows

8 matrices = 1 SoA ‘package’

optimal: contiguous load/store
Interdisciplinary Workflow

- ODEs
- PDEs
- Graphs
- ... 

High-Level Prototype (Mathematica)
- domain scientist’s tool
- high level
- symbolic solvers
- arbitrary precision
- very limited performance

OpenCL kernel within Mathematica
- replace some code with OpenCL
- compare results
- figure out numerics
- use accelerators in MM

Mathematical Model

C++ App. with OpenCL kernel
- start single node
- OpenCL 1.2 for hotspots
- modern C++ 11/14/17
- CMake for building

Distributed Application (MPI)
Interdisciplinary Workflow

Mathematical Model
- ODEs
- PDEs
- Graphs
- ...

High-Level Prototype (Mathematica)
- domain scientist’s tool
- high level
- symbolic solvers
- arbitrary precision
- very limited performance

OpenCL kernel within Mathematica
- replace some code with OpenCL
- compare results
- figure out numerics
- use accelerators in MM

C++ App. with OpenCL kernel
- start single node
- OpenCL 1.2 for hotspots
- modern C++ 11/14/17
- CMake for building

Distributed Application (MPI)
- scale to multiple nodes
- partitioning,
- neighbour exchange, ...
- wrapped MPI 3.0
Partitioning the Hierarchy

- **Problem**: map hierarchy graph nodes to $n$ partitions (compute nodes)
  - minimise communication
  - minimise load imbalance

$\Rightarrow$ GP is **NP-hard**

$\Rightarrow$ hierarchy graph is highly connected
Partitioning the Hierarchy

- **Problem**: map hierarchy graph nodes to $n$ partitions (compute nodes)
  - minimise communication
  - minimise load imbalance

⇒ GP is **NP-hard**
⇒ hierarchy graph is highly connected
Partitioning the Hierarchy

- **Problem**: map hierarchy graph nodes to \( n \) partitions (compute nodes)
  - minimise communication
  - minimise load imbalance

⇒ GP is **NP-hard**

⇒ hierarchy graph is highly connected
Partitioning the Hierarchy

- **Problem**: map hierarchy graph nodes to $n$ partitions (compute nodes)
  - minimise communication
  - minimise load imbalance

$\Rightarrow$ GP is **NP-hard**

$\Rightarrow$ hierarchy graph is highly connected
Partitioning the Hierarchy (PS-I with $d = 3$)

- **METIS** generated partitionings
  - partitioning quality mostly resilient to METIS settings

\[
\begin{array}{cccc}
16 & 15 & 15 & 15 \\
9803 & 91.7\% & 12508 & 127.6\% \\
128 & 127 & 125 & 127 \\
1225 & 98.0\% & 2958 & 241.4\% \\
256 & 241 & 212 & 255 \\
613 & 98.8\% & 1689 & 274.8\% \\
512 & 281 & 124 & 414 \\
306 & 99.0\% & 929 & 303.3\% \\
\end{array}
\]
Partitioning the Hierarchy (PS-I with $d = 3$)

- **METIS** generated partitionings
  - partitioning quality mostly resilient to METIS settings

<table>
<thead>
<tr>
<th>partitions</th>
<th>neighbor parts.</th>
<th>hierarchy nodes per part. (avg)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>avg</td>
<td>min</td>
</tr>
<tr>
<td>16</td>
<td>15</td>
<td>15</td>
</tr>
<tr>
<td>32</td>
<td>31</td>
<td>31</td>
</tr>
<tr>
<td>64</td>
<td>62</td>
<td>60</td>
</tr>
<tr>
<td>128</td>
<td>127</td>
<td>125</td>
</tr>
<tr>
<td>256</td>
<td>241</td>
<td>212</td>
</tr>
<tr>
<td>512</td>
<td>281</td>
<td>124</td>
</tr>
</tbody>
</table>
### Partitioning the Hierarchy (PS-I with $d = 3$)

- **METIS** generated partitionings
  - partitioning quality mostly resilient to METIS settings

<table>
<thead>
<tr>
<th>partitions</th>
<th>neighbor parts.</th>
<th>hierarchy nodes per part. (avg)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>avg</td>
<td>min</td>
</tr>
<tr>
<td>16</td>
<td>15</td>
<td>15</td>
</tr>
<tr>
<td>32</td>
<td>31</td>
<td>31</td>
</tr>
<tr>
<td>64</td>
<td>62</td>
<td>60</td>
</tr>
<tr>
<td>128</td>
<td>127</td>
<td>125</td>
</tr>
<tr>
<td>256</td>
<td>241</td>
<td>212</td>
</tr>
<tr>
<td>512</td>
<td>281</td>
<td>124</td>
</tr>
</tbody>
</table>

⇒ highly connected partitioning graph  
⇒ almost **all-to-all**
Partitioning the Hierarchy (PS-I with $d = 3$)

- **METIS** generated partitionings
  - partitioning quality mostly resilient to METIS settings

<table>
<thead>
<tr>
<th>partitions</th>
<th>neighbor parts.</th>
<th>hierarchy nodes per part. (avg)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>avg</td>
<td>min</td>
</tr>
<tr>
<td>16</td>
<td>15</td>
<td>15</td>
</tr>
<tr>
<td>32</td>
<td>31</td>
<td>31</td>
</tr>
<tr>
<td>64</td>
<td>62</td>
<td>60</td>
</tr>
<tr>
<td>128</td>
<td>127</td>
<td>125</td>
</tr>
<tr>
<td>256</td>
<td>241</td>
<td>212</td>
</tr>
<tr>
<td>512</td>
<td>281</td>
<td>124</td>
</tr>
</tbody>
</table>

⇒ almost all nodes required by other partitions
⇒ prevents inner/outer communication/computation overlap
Partitioning the Hierarchy (PS-I with $d = 3$)

- **METIS** generated partitionings
  - partitioning quality mostly resilient to METIS settings

<table>
<thead>
<tr>
<th>partitions</th>
<th>neighbor parts.</th>
<th>hierarchy nodes per part. (avg)</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>avg</td>
<td>min</td>
<td>max</td>
<td>nodes</td>
<td>shared %</td>
<td>halo</td>
<td>overhead %</td>
</tr>
<tr>
<td>16</td>
<td>15</td>
<td>15</td>
<td>15</td>
<td>9803</td>
<td>91.7%</td>
<td>12508</td>
<td>127.6%</td>
</tr>
<tr>
<td>32</td>
<td>31</td>
<td>31</td>
<td>31</td>
<td>4902</td>
<td>95.0%</td>
<td>7858</td>
<td>160.3%</td>
</tr>
<tr>
<td>64</td>
<td>62</td>
<td>60</td>
<td>63</td>
<td>2451</td>
<td>96.8%</td>
<td>4739</td>
<td>193.2%</td>
</tr>
<tr>
<td>128</td>
<td>127</td>
<td>125</td>
<td>127</td>
<td>1225</td>
<td>98.0%</td>
<td>2958</td>
<td>241.4%</td>
</tr>
<tr>
<td>256</td>
<td>241</td>
<td>212</td>
<td>255</td>
<td>613</td>
<td>98.8%</td>
<td>1689</td>
<td>274.8%</td>
</tr>
<tr>
<td>512</td>
<td>281</td>
<td>124</td>
<td>414</td>
<td>306</td>
<td>99.0%</td>
<td>929</td>
<td>303.3%</td>
</tr>
</tbody>
</table>

⇒ more halo nodes than local nodes
Distributed Memory HEOM: Multi-Node C++ Application

- Simple transition to multi-node application
- Partition Mapping as additional input (METIS)
- Generated Hierarchy Partition as new input for Instance
  ⇒ Instance and Solver do not care
- Communicator encapsulates MPI-3
- Neighborhood Collectives
- Derived Data Types
  ⇒ fully declarative communication API
- ODE can trigger action
  ⇒ neighbor exchange prior to evaluation
Distributed Memory HEOM: Multi-Node C++ Application

- simple transition to multi-node application

- Partition Mapping as additional input (METIS)
- generated Hierarchy Partition as new input for Instance
  ⇒ Instance and Solver do not care

- Communicator encapsulates MPI-3
- Neighborhood Collectives
- Derived Data Types ⇒ fully declarative communication API
- ODE can trigger action ⇒ neighbor exchange prior to evaluation

HEOM Config → HEOM Configuration → Hierarchy Graph → Hierarchy Partition → Instance → ODE

OpenCL Config → Solver

Results
Distributed Memory HEOM: Multi-Node C++ Application

- simple transition to multi-node application
- **Partition Mapping** as additional input (METIS)
Distributed Memory HEOM: Multi-Node C++ Application

- simple transition to multi-node application
- *Partition Mapping* as additional input (METIS)
- generated *Hierarchy Partition* as new input for *Instance*
  ⇒ Instance and Solver do not care
Distributed Memory HEOM: Multi-Node C++ Application

- simple transition to multi-node application
- **Partition Mapping** as additional input (METIS)
- generated **Hierarchy Partition** as new input for **Instance**
  - Instance and Solver do not care
- **Communicator** encapsulates MPI-3
  - Neighborhood Collectives
  - Derived Data Types
  - fully **declarative** communication API

- ODE can trigger action
  - neighbor exchange prior to evaluation

---

**HEOM Config**
- HEOM Configuration
- Hierarchy Graph
- Hierarchy Partition
- Instance
- Communicator
- OpenCL Config
- Solver
- ODE
- register neighbor exchange action
- Results
Distributed Memory HEOM: Multi-Node C++ Application

- simple transition to multi-node application
- **Partition Mapping** as additional input (METIS)
- generated *Hierarchy Partition* as new input for *Instance*
  ⇒ Instance and Solver do not care
- **Communicator** encapsulates MPI-3
  - Neighborhood Collectives
  - Derived Data Types
  ⇒ fully **declarative** communication API
- **ODE** can trigger action
  ⇒ neighbor exchange prior to evaluation
Interdisciplinary Workflow

Mathematical Model
- ODEs
- PDEs
- Graphs
- ...

High-Level Prototype (Mathematica)
- domain scientist’s tool
- high level
- symbolic solvers
- arbitrary precision
- very limited performance

OpenCL kernel within Mathematica
- replace some code with OpenCL
- compare results
- figure out numerics
- use accelerators in MM

C++ App. with OpenCL kernel
- start single node
- OpenCL 1.2 for hotspots
- modern C++ 11/14/17
- CMake for building

Distributed Application (MPI)
- scale to multiple nodes
- partitioning,
- neighbour exchange, ...
- wrapped MPI 3.0

Optimisation / Production Runs
Interdisciplinary Workflow

Mathematical Model
- ODEs
- PDEs
- Graphs
- ... 

High-Level Prototype (Mathematica)
- domain scientist’s tool
- high level
- symbolic solvers
- arbitrary precision
- very limited performance

OpenCL kernel within Mathematica
- replace some code with OpenCL
- compare results
- figure out numerics
- use accelerators in MM

C++ App. with OpenCL kernel
- start single node
- OpenCL 1.2 for hotspots
- modern C++ 11/14/17
- CMake for building

Distributed Application (MPI)
- scale to multiple nodes
- partitioning,
  neighbour exchange, ...
- wrapped MPI 3.0

Optimisation / Production Runs
- always collect perf. data
- profile/tune code
- explore new architectures
Benchmarks: Work-item Granularity

Impact of Work-item Granularity

average solver step runtime [ms]

Granularity

Matrix
Element
Benchmarks: Work-item Granularity

Impact of Work-item Granularity

Granularity

Matrix
Element

Impact of Work-item Granularity

average solver step runtime [ms]

fmo_22baths_d3.cfg

lhcii_1bath_d8.cfg

⇒ CPUs: 1.2× to 1.35× speedup for Matrix granularity
Benchmarks: Work-item Granularity

Impact of Work-item Granularity

⇒ GPUs: up to 6.7× (K40) and 7.2× (W8100) speedup for Element granularity
Benchmarks: Memory Layout

Impact of Configurable Memory Layout

<table>
<thead>
<tr>
<th>Layout</th>
<th>AoS</th>
<th>AoSoA</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 SKL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2 HSW</td>
<td></td>
<td></td>
</tr>
<tr>
<td>KNL</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

For the configurations:
- fmo_22baths_d3.cfg
- lhcii_1bath_d8.cfg

The average solver step runtime is shown in milliseconds for different layouts and hardware configurations.
Benchmarks: Memory Layout

Impact of Configurable Memory Layout

<table>
<thead>
<tr>
<th>Layout</th>
<th>AoS</th>
<th>AoSoA</th>
</tr>
</thead>
<tbody>
<tr>
<td>fmo_22baths_d3.cfg</td>
<td>1500</td>
<td>500</td>
</tr>
<tr>
<td>lhcii_1bath_d8.cfg</td>
<td>2000</td>
<td>1000</td>
</tr>
</tbody>
</table>

⇒ SKL and HSW: 1.3× to 2.4× speedup with AoSoA
Benchmarks: Memory Layout

Impact of Configurable Memory Layout

- fmo_22baths_d3.cfg
- lhcii_1bath_d8.cfg

average solver step runtime [ms]

<table>
<thead>
<tr>
<th>Layout</th>
<th>fmo_22baths_d3.cfg</th>
<th>lhcii_1bath_d8.cfg</th>
</tr>
</thead>
<tbody>
<tr>
<td>AoS</td>
<td>2 SKL</td>
<td>2 SKL</td>
</tr>
<tr>
<td>AoSoA</td>
<td>2 HSW</td>
<td>2 HSW</td>
</tr>
<tr>
<td></td>
<td>KNL</td>
<td>KNL</td>
</tr>
</tbody>
</table>

⇒ KNL: 1.6× to 2.8× speedup with AoSoA
Benchmarks: Performance Portability

Performance Portability Relative to Xeon (SKL)

<table>
<thead>
<tr>
<th>Hardware</th>
<th>fmo_22baths_d3.cfg</th>
<th>lhci_1bath_d8.cfg</th>
</tr>
</thead>
<tbody>
<tr>
<td>2x Xeon (SKL)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2x Xeon (HSW)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ex. from FLOPS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Xeon Phi (KNL)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ex. from FLOPS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tesla K40</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ex. from FLOPS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FirePro W8100</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ex. from FLOPS</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

average solver step runtime [ms]
Benchmarks: Performance Portability

Performance Portability Relative to Xeon (SKL)

Average solver step runtime [ms]

Hardware
- 2x Xeon (SKL)
- 2x Xeon (HSW)
- ex. from FLOPS
- Xeon Phi (KNL)
- ex. from FLOPS
- Tesla K40
- ex. from FLOPS
- FirePro W8100
- ex. from FLOPS

⇒ SKL (Xeon) is the reference
Benchmarks: Performance Portability

Performance Portability Relative to Xeon (SKL)

average solver step runtime [ms]

⇒ gray bars are **expected runtimes** extrapolated from peak FLOPS
Benchmarks: Performance Portability

Performance Portability Relative to Xeon (SKL)

⇒ Older Haswell Xeon exceeds expectations, due to better OpenCL support
Performance Portability Relative to Xeon (SKL)

average solver step runtime [ms]

⇒ Good: within 30 % of expectation
Benchmarks: Performance Portability

⇒ KNL and K40 sensitive to **irregular accesses** from extreme coupling in this scenario.
Benchmarks: Scalability

Strong Scaling of PS I with 3 Layers

- **Runners** [ms]
- **Nodes**: 16 32 64 128 256 512
- **Speedup vs. 16 nodes**

- **Communication**
- **Compute**
- **Sum**
Benchmarks: Scalability

Strong Scaling of PS I with 3 Layers

⇒ communication cost > computation cost
⇒ better node utilisation means worse scalability
Benchmarks: Scalability

Strong Scaling of PS I with 3 Layers

- Communication scales ideally
- Compute scales ideally
- Sum scales ideally

⇒ compute scales ideally
Benchmarks: Scalability

Strong Scaling of PS I with 3 Layers

- all-to-all like neighbour-exch. limits scalability

⇒ all-to-all like neighbour-exch. limits scalability

communication  compute  sum
Benchmarks: Scalability

Strong Scaling of PS I with 3 Layers

runtime [ms]
16 32 64 128 256 512

speedup vs. 16 nodes

⇒ 14.1× speedup
16 → 512 nodes

communication compute sum
Summary

Lessons’s learned:

- **interdisciplinary workflow** is key for developing HPC codes
- standards (OpenCL, MPI-3, ...) enable **portability**
- a **flexible** design enables **portable performance**
  - ⇒ leverage **runtime compilation**
  - ⇒ **work-item granularity** and **memory and layout**

DM-HEOM:

- first **Distributed Memory HEOM** implementation
- pushes the boundary of feasible problem sizes
- **practical scalability** from laptops to supercomputers
Summary

Lessons’s learned:

- ***interdisciplinary workflow*** is key for developing HPC codes
- standards (OpenCL, MPI-3, ...) enable **portability**
- a **flexible** design enables **portable performance**
  - ⇒ leverage **runtime compilation**
  - ⇒ **work-item granularity** and **memory and layout**

DM-HEOM:

- first **Distributed Memory HEOM** implementation
- pushes the boundary of **feasible problem sizes**
- **practical scalability** from laptops to supercomputers
Thank you.

Feedback? Questions? Ideas?

noack@zib.de

This work was funded by the Deutsche Forschungsgemeinschaft (DFG), projects RE 1389/8-1 and KR 2889/7-1. Additional funding came from Intel Corp. within the activities of the "Research Center for Many-Core HPC" at ZIB, an Intel Parallel Computing Center. The authors acknowledge the North-German Supercomputing Alliance (HLRN) for providing computing time on the Cray XC40.