Accelerating Octo-Tiger: Stellar Mergers on Intel Knights Landing with HPX

David Pfander*, Gregor Daiß*, Dominic Marcello**, Hartmut Kaiser**, Dirk Pflüger*
* University of Stuttgart ** Louisiana State University

May 14, 2018
Agenda

HPX and Vc

Stellar Mergers with Octo-Tiger

A Futurized Fast Multipole Method

Interactions Stencil Kernels

Results on Intel Knights Landing (and Skylake)

Conclusion and Outlook
HPX for Parallelization

• HPX is a modern C++-based parallelization framework:

Same syntax and semantics locally and distributed
• Supports futurization-continuation model
• Work stealing scheduler
Vc for Vectorization

using Vc::double_v;

double_v res(0.0);
for (size_t i = 0; i < N; i++) {
    double_v a(&A[i], flags::element_aligned);
    double_v b(&B[i], flags::element_aligned);
    res += a * b; // FMA (expr. templates)
} // now reduce res

- Scalar-like vector expressions
- Portable (SSE, AVX1/2/512, NEON)
- Prescriptive vectorization (cf. OpenMP SIMD clause)
- (Developed by Matthias Kretz, proposed for standardization (P0214R3))
Stellar Mergers with Octo-Tiger

- Simulates star systems with two or more stars
- Important scenario: merger in double white dwarf systems
- Stellar merger lead to astrophysical phenomena such as type Ia supernovae
- (Written in C++11/14)
Stellar Mergers with Octo-Tiger

- Simulates star systems with two or more stars
- Important scenario: merger in double white dwarf systems
- Stellar merger lead to astrophysical phenomena such as type Ia supernovae
- (Written in C++11/14)
Stellar Mergers in 3d

- Unique: Octo-Tiger conserves angular (and linear) momentum
Calculating Gravity

- Fast multipole method (FMM) to compute gravity
- Adaptive octrees as main data structure (AMR)
- Nodes have $8 \times 8 \times 8$ subgrid
- History of fast tree-based codes (Dehnen 2000)
Parallelizing Octo-Tiger with HPX

- Octo-Tiger uses $8 \times 8 \times 8$ subgrids as parallelization primitives
- Work stealing by moving octree nodes
- Octree nodes are AGAS objects
Futurized Tree Traversals

(1) Calc. multipole moments
- (2) most expensive!
- Steps partially overlap, thanks to futurization and HPX!
- Scalability proven with full-system runs (9640 KNLs) on Cori at NERSC

(2) Same-level interactions

(3) Apply multipole expansions

David Pfander, Gregor Daiß, Dominic Marcello, Hartmut Kaiser and Dirk Pflüger: Accelerating Octo-Tiger: Stellar Mergers on Intel Knights Landing with HPX
May 14, 2018
Multipole Interaction Stencil

- Opening criterion:

\[
\frac{1}{|Z_1^f - Z_2^f|} \leq \Theta < \frac{1}{|Z_1^c - Z_2^c|}
\]

- \(Z_1^f\) and \(Z_2^f\) coordinate centers of multipoles, \(Z_1^c\) and \(Z_2^c\) coordinate center of parents
- \(\Theta := 0.35\) \(\Rightarrow\) stencil with 1074 elements
- Need access to neighbor’s subgrids (27 element neighborhood)
Multipole Expansion Calculation

- Calculation of multipole expansion coefficients $L_q^{(i)}$ for cell $q$
- (1) Calculate $D_0^{(0)}, D_m^{(1)}, D_{mn}^{(2)}, D_{mnp}^{(3)}$ (114 FLOPS each), (2) Calculate $L_q^{(i)}$:

  $$L_q^{(0)} := \sum_l \left[ M_l^{(0)} D_l^{(0)}(R_l,q) + M_{l,m}^{(1)} D_m^{(1)}(R_l,q) + M_{l,mn}^{(2)} D_{mn}^{(2)}(R_l,q) + M_{l,mnp}^{(3)} D_{mnp}^{(3)}(R_l,q) \right]$$

  $$L_q^{(1)} := \sum_l \left[ M_l^{(0)} D_m^{(1)}(R_l,q) + M_{l,n}^{(1)} D_{mn}^{(2)}(R_l,q) + M_{l,np}^{(2)} D_{mnp}^{(3)}(R_l,q) \right]$$

  $$L_q^{(2)} := \sum_l \left[ M_l^{(0)} D_{mn}^{(2)}(R_l,q) + M_{l,p}^{(1)} D_{mnp}^{(3)}(R_l,q) \right]$$

  $$L_q^{(3)} := \sum_l M_l^{(0)} D_{mnp}^{(3)}(R_l,q)$$

  (Multipole moments $M^{(i)}$ given)

- $l$ iterates neighboring cells through stencil (1074 elements)
- $m, n, p$ iterated in $\{0, 1, 2\} \implies 40$ coefficients (20 unique)
Vector-Friendly Storage

- Small arrays per cell in subgrid, for $L^{(i)}$:

- Implemented as Struct-of-Array for vectorization:

- For $L^{(i)}$, $M^{(i)}$, $D^{(i)}$: 20 + 20 + 35 = 75 coefficients:
  - Per cell in subgrid
  - Loaded for each stencil element
  - Scalar-like branch-free formulation of FMM kernels possible
### Obtained FMM Compute Kernels

<table>
<thead>
<tr>
<th>kernel</th>
<th>FLOPs per interaction</th>
<th>stencil size</th>
<th>floating point op.</th>
<th>Mem.</th>
<th>arith. intensity</th>
</tr>
</thead>
<tbody>
<tr>
<td>M2M/M2P corr.</td>
<td>455</td>
<td>1,074</td>
<td>173,089,280</td>
<td>920 kB</td>
<td>183 F/B</td>
</tr>
<tr>
<td>M2M/M2P</td>
<td>295</td>
<td>1,074</td>
<td>112,222,720</td>
<td>896 kB</td>
<td>122 F/B</td>
</tr>
<tr>
<td>P2P/P2M corr.</td>
<td>360</td>
<td>1,074</td>
<td>136,949,760</td>
<td>872 kB</td>
<td>153 F/B</td>
</tr>
<tr>
<td>P2P/P2M</td>
<td>215</td>
<td>1,074</td>
<td>81,789,440</td>
<td>752 kB</td>
<td>106 F/B</td>
</tr>
</tbody>
</table>

- Four kernels:
  - Less work needed for leaf nodes
  - Angular correction
- M2P to be read as Multipole-to-Monopole (“Particle”)
- Generally compute bound, large caches needed
Experiments

Intel Xeon Phi 7250 (KNL):
- 68C, 1MB L2 cache per dual-core tile
- 1.4 GHz base, 1.32 GHz under heavy AVX512 load
- 2872 GFLOPS double precision peak

2xIntel Xeon Silver 4116 (SKL):
- 2x12C, 1 MB L2 cache per core
- 2.1 GHz base, 1.5 GHz under heavy AVX512 load
- 576 GFLOPS double precision peak

Evaluation scenario:
- Rotating star in equilibrium
- Domain 30x larger than star, adaptively-refined grid
- Demonstrates angular momentum conservation
Node-Level Scaling

- Calculated with level 6 grid (3081 nodes), whole application!
- Cache too small on KNL
FMM Kernels Performance

- GFLOPS average over all four kernel variants
- KNL needs a larger grid
- 14% peak on KNL, 34% peak on SKL, KNL faster
Application Runtime

- Strong improvements for FMM
- Other application parts need to be further improved (regridding)
Conclusion and Outlook

Lessons learned:
• Futurization-continuation scaled up to 9640 nodes
• HPX+Vc enables fast and scaling high-level impl.
• KNL is a tough target for node-level performance

Next Steps:
• More cache-friendly FMM algorithms
• Deal with non-FMM (grid-related) bottlenecks
• Porting to Piz Daint at CSCS (Nvidia P100)
Conclusion and Outlook

Lessons learned:
- Futurization-continuation scaled up to 9640 nodes
- HPX+Vc enables fast and scaling high-level impl.
- KNL is a tough target for node-level performance

Next Steps:
- More cache-friendly FMM algorithms
- Deal with non-FMM (grid-related) bottlenecks
- Porting to Piz Daint at CSCS (Nvidia P100)

Questions?
Computing the Gradients of the Gravitational Potential

- $X, Y \in \mathbb{R}^3$ center of masses, with $R := X - Y$, $d := ||R||_2$
- Gradients of the gravitational potential are
  \[
  \nabla^{(n)} g(R) = -\nabla^{(n)} \frac{1}{d} := D^{(n)}
  \]
- Thus,
  \[
  D^{(0)}(R) := -\frac{1}{d},
  \]
  \[
  D^{(1)}_i(R) := \frac{R_i}{d^3},
  \]
  \[
  D^{(2)}_{ij}(R) := -\frac{3 R_i R_j - \delta_{ij} d^2}{d^5},
  \]
  \[
  D^{(3)}_{ijk}(R) := \frac{15 R_i R_j R_k - 3 (\delta_{ij} R_k + \delta_{jk} R_i + \delta_{ki} R_j) d^2}{d^7}
  \]
- Iterating $i, j, k \in \{0, 1, 2\}$, 40 expression to evaluate (20 unique)
- Omitted $D^{(4)}$
Early Result on P100

**Influence of the Cuda Streams on the Runtime**

- Runtime using one NVIDIA Tesla P100
- Runtime using two NVIDIA Tesla P100

**Distributed Scaling on Piz Daint with and without GPUs**

- Parallel efficiency using both CPU and GPU
- Parallel efficiency using only CPU

**Key Points**

- Using GPU as pure Co-processor through CUDA stream
- Implementation through HPX compute + CUDA clang
- Runtime improvements despite somewhat starving GPU