Density Functional Theory for Condensed Matter Research

October 12, 2015 | R. Zeller P. Baumeister T. Hater D. Pleiter | Juelich Supercomputing Center
Density functional theory

Dirac, Hohenberg-Kohn: electronic density can be used instead of many-electron wavefunctions

"At a fundamental level, DFT can be used to describe all of chemistry, biochemistry, biology, nanosystems and materials. Everything in our terrestrial world depends on the motions of electrons – therefore, DFT literally underlies everything."

as Axel Becke says in the Nature feature article: THE TOP 100 PAPERS (2014)
Computational bottleneck

- conventional DFT methods require $O(N^3)$ computing time
- a few thousand atoms can be treated today
- what can be gained by 1000fold increase in CPU power?
- $N$, number of atoms, can increase by a factor of 10
- system diameter can increase by a factor $10^{1/3} \approx 2$
Linear scaling methods

- Motivation: reduce scaling to $O(N^2)$ to $O(N)$
- Approach: trade in accuracy for speed

KKRnano: code developed at IAS and PGI (FZ Jülich)
Aim 1: keep precision as much as possible
Aim 2: support massively parallel computing

Rudolf Zeller, Alexander Thieß, Elias Rabel, Stefan Blügel, Peter Dederichs, Matthias Bolten (Wuppertal), Irad Yavneh (Haifa), Marcel Bornemann, Roman Kovacic, Masako Ogura, Paul Baumeister(JSC), Thorsten Hater (JSC)
Applications of KKRnano

Ongoing projects
- Phase-Change Materials
  - GeSbTe disorder & localization
- Dilute Magnetic Semiconductors
  - GaN:Gd colossal magnetic moments
- Martensitic Phase Transition
  - NiTi strain glass
- Transition Metal Oxides
  - SrTiO3 resistive switching

Published projects
Methodology of KKRnano

- conventional DFT methods: \[ -\nabla_\mathbf{r}^2 + V(\mathbf{r}) \varphi_i(\mathbf{r}) = \epsilon_i \varphi_i(\mathbf{r}) \]

- KKRnano: \[ -\nabla_\mathbf{r}^2 + V(\mathbf{r}) - \epsilon \] \[ G(\mathbf{r}, \mathbf{r}', \epsilon) = \delta(\mathbf{r} - \mathbf{r}') \], but as

\[
G(\mathbf{r}, \mathbf{r}', \epsilon) = g(\mathbf{r}, \mathbf{r}', \epsilon) + \int d\mathbf{r}'' g(\mathbf{r}, \mathbf{r}'', \epsilon) [V(\mathbf{r}'') - V^\text{ref}(\mathbf{r}'')] G(\mathbf{r}'', \mathbf{r}', \epsilon)
\]

multiple scattering theory: Lord Rayleigh (1892)

KKR method: Korringa (1947), Kohn, Rostoker (1954)

\[
G(\mathbf{r} + \mathbf{R}_n^{\mathbf{r}}, \mathbf{r}' + \mathbf{R}_n'^{\mathbf{r}'}; \epsilon) = \delta_{nn'} G^n_s(\mathbf{r}, \mathbf{r}'; \epsilon) + \sum_{LL'} R^n_L(\mathbf{r}; \epsilon) G^n_{LL'}(\epsilon) R^n_{L'}(\mathbf{r}'; \epsilon)
\]
Methodology of KKRnano

- divide space into cells
- solve single-cell problems
  \[ G^n_S(r, r'; \epsilon), R^n_L(r; \epsilon), t^n''_{LL'}(\epsilon) \]
  computing effort \( O(N) \)
- solve matrix equation
  by iteration \( \Rightarrow O(N^2) \)
  by truncation \( \Rightarrow O(N) \)

\[
G^{nn'}_{LL'}(\epsilon) = g^{nn'}_{LL'}(\epsilon) + \sum_{n'' L'' L''' } g^{n'n''}_{L'L''}(\epsilon) \Delta t^{n''}_{L'L''}(\epsilon) G^{n''n'}_{L''L'}(\epsilon)
\]
Parallel efficiency

KKRnano is member of the High Q-Club, codes that can utilise the entire 28-rack BlueGene/Q system at JSC.

Strong-scaling obtained for increasing number of OpenMP threads and associated decreasing number of MPI tasks per compute node.
Performance Characterisation of KKRnano

- Main contribution: QMR iterative solver
- 90% of runtime for large $N_{\text{Atom}}$, especially in linear scaling
- Linear scaling regime from around $N_{\text{Atom}} \sim \mathcal{O}(1000)$
- Block sparse operator over $\mathbb{C}$

![Diagram showing block sparse operator over $\mathbb{C}$]
KKRnano on POWER8 and CUDA

- Baseline code has been ported unaltered to POWER8
- Task arithmetic intensity $4^{Flop/B}$
- All computations in double precision

<table>
<thead>
<tr>
<th>Component</th>
<th>Bandwidth</th>
<th>DP Ops</th>
<th>Limit</th>
</tr>
</thead>
<tbody>
<tr>
<td>P8 (20 cores)</td>
<td>$&gt; 300^{GB/s}$</td>
<td>$0.56^{TF/s}$</td>
<td>$0.56^{TF/s}$</td>
</tr>
<tr>
<td>K40 (1 GPU)</td>
<td>$&lt; 280^{GB/s}$</td>
<td>$1.6^{TF/s}$</td>
<td>$\sim 1^{TF/s}$</td>
</tr>
<tr>
<td>K80 (1 Chip)</td>
<td>$&lt; 240^{GB/s}$</td>
<td>$1.4^{TF/s}$</td>
<td>$\sim 1^{TF/s}$</td>
</tr>
</tbody>
</table>
Performance Results: Baseline KKRnano
A mini-application

- Stand-alone mini-app for QMR solver
  - C++ host version using OpenMP
  - CUDA 7.0 using cuSparse and custom kernels
- Full solver on the GPU
  - Data transfer once per solver step
  - In $\sim 55\text{KiB} \cdot N_{\text{Atom}}$
  - Out $\sim 4\text{KiB} \cdot N_{\text{Atom}}$
Benchmark

- From small scale runs well in to linear scaling \(100 \leq N_{\text{Atom}} \leq 4000\)
- Set number of QMR iterations to 1000, typical \(O(100)\)

\[
\sqrt[bszmm]{2 \cdot 1000 \cdot 8 \cdot 16^3 \cdot 13 \cdot N_{\text{Atom}}} \text{ DP-Flops}
\]

- POWER8
  - 20 cores with SMT2, best out of 1-160 threads on 20 cores
- K40
  - Single device, one process (clocks: 875/3004MHz)
- All GPU timings include the data transfer times
Performance Results: QMR solver

- ATLAS \texttt{zgemm} 94\% on CPU
- \texttt{cuSparse bszmm} 82\% on GPU

Thread Scaling
Analysis of CPU variant

- Less than expected fraction of available performance
  - 8% using four threads per core (shortest time to solution)
  - 13% of core peak at single thread

- Performance counters: stall cycles and instruction mix
  - Almost no vectorised instructions
    - Misses roughly one half of available performance
  - Almost no cache misses (Good)
  - Main stall reasons are inter-instruction dependencies (RAW)
Focus on the Block Sparse Multiplication

- Split out the CPU part into separate, smaller benchmark
- No significant difference between ATLAS and handwritten
- Real and imaginary parts in separate arrays
  - structure-of-arrays (SoA) instead of arrays-of-structures (AoS)
    - Vastly improves vectorisation
- Achieved fraction of peak better than cuSparse
  - 25% single threaded / 42% SMT4
  - Stall reasons and fractions are similar to before: RAW
  - Very long (34 elements) linear chain of assembly instructions
- 0.96 instructions per cycle out of four (2x ld + 2x fmadd)
  - Not limited by bandwidth or issue ports
  - OOO execution does not rescue us here
  - Issue: long dependency chain of fmadd
And one more step

- Re-ordering the loops, tune cache blocking
- Less SIMD instructions, but
  - 42% ST / 50% SMT4
  - Similar results between GCC and XLC
- Scales up to the full node: $280^{GF/s}$
- A single GPU reaches $330^{GF/s}$
Results

- One POWER8 core: $28^{GF/s}$, scales to full node
- Best time to solution with SMT4
Conclusion CPU Implementation

- SoA format for complex numbers is a significant gain for the matrix multiplication
- Fraction of single core peak achieved
  - 42% ST / 50% SMT4
  - Full node can compete with one GPU
- Reordering the matrix elements necessary
  - AoS → SoA
  - Transposing each block
  - Not included in timings
Back to the GPU

- Both GPUs running at peak clock of 875MHz
- Analysed using `nvprof` and `nvvp`
- Performance limiter: shared memory bandwidth
- Experiments with SoA format proved to be significantly worse
K40 vs K80 (one GK 210)
Multiple Host processes per GPU

- One host process does not exhaust the resources of the GPU
  - Neither memory capacity nor compute
- Using the Multiple Process Service (mps) facility to share GPUs between host processes
- An obvious sweet spot would be one atom per core
  - One MPI task per core results in 10 processes per GPU
- Investigate scaling with processes per K40 GPU
Results with mps

Parallel efficiency \( \epsilon = n \cdot \frac{T_{\text{ser}}}{T_{\text{par}}} \)

October 12, 2015 T. Hater P. Baumeister D. Pleiter R. Zeller
Conclusion GPU implementation

- Utilising GPU computing for the solver results in further speed-up compared to the improved CPU baseline
- Multiple tasks allow for efficient use of GPU resources
- Extrapolated whole application speed-ups

one node vs one GPU

<table>
<thead>
<tr>
<th></th>
<th>Base</th>
<th>SoA</th>
<th>Tuned</th>
<th>K80/2</th>
<th>K40</th>
<th>w/ mps</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>2.3</td>
<td>3.4</td>
<td>4.2</td>
<td>4.4</td>
<td>4.8</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1.8</td>
<td>1.9</td>
<td>2.0</td>
<td>2.3</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1.0</td>
<td>1.1</td>
<td>1.3</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1.1</td>
<td>1.2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td></td>
<td>1.1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Base SoA Tuned K80/2 K40 w/ mps
Planned: GPU/CPU Cooperation

- The work is naturally divided into independent tasks
- Processing a row of the sparse operator constitutes a task
  - CPU needs to additionally manage data transfers
  - Use async progress to hide behind CPU computation
  - Push rows to GPU, pull result blocks back
- Split tasks according to available resources
  - Take into account efficiencies
  - Flexible, forwards compatible
- Current configuration and efficiencies
  - 2 GPUs: \(0.2 \times 1.6\,\text{TF/s} \approx 0.64\,\text{TF/s}\)
  - 2 CPUs: \(0.5 \times 0.56\,\text{TF/s} \approx 0.28\,\text{TF/s}\)
  - Ratio \(\approx 2.3\)
Summary

- We ported an essential part of KKRnano to POWER8 + GPU
- Several significant improvements by tuning the baseline code
  - Improved storage format, data re-ordering
  - Efficient use of hardware: SIMD, caches and NUMA
- The full solver was ported to the GPU
  - Data format remains largely identical
  - Exploit cuSparse for efficient block sparse multiplication
  - Utilise mps to improve efficiency
- Future work
  - Solvers can be easily plugged into default KKRnano
  - Explore CPU/GPU collaboration in code
  - Understand/Develop performance of the GPU code
Questions?
Experimental Parameters

- Alignment to cache lines
- OpenMP enabled benchmarks
  - thread pinning
  - first touch allocation
- CUDA 7.0

<table>
<thead>
<tr>
<th>Count</th>
<th>Clock (MHz)</th>
<th>Compiler</th>
</tr>
</thead>
<tbody>
<tr>
<td>P8 20 Cores</td>
<td>3650</td>
<td>GCC 4.9 -Ofast -flto</td>
</tr>
<tr>
<td>P8 20 Cores</td>
<td>3650</td>
<td>XLC 13$\beta$ -05</td>
</tr>
<tr>
<td>K40 1 GPU</td>
<td>875/2505</td>
<td>-02 -use_fast_math</td>
</tr>
<tr>
<td>K80 1 Chip</td>
<td>875/3004</td>
<td>-02 -use_fast_math</td>
</tr>
</tbody>
</table>
Dependency chains

- Two chains of 34 instructions
- $\sim 0.5$ vxfmadd per cycle
- FP pipeline latency exposed, despite OOO
- SMT helps a bit
Instruction mix

- No significant change
- More scalar stores in the newer version
Stall cycles breakdown

- Consequence: shifted some stalls from VSU to LSU
- Still, overall gain in terms of efficiency