Code optimization for Cell/B.E.

Opportunities for ABINIT – a software package for physicists

Timo Schneider[1], Simon Wunderlich[1], Wolfgang Rehm[1], Torsten Hoefler[1,2], Heiko Schick[3]
[1] Chemnitz University of Technology, Germany [2] Indiana University, USA [3] IBM Deutschland Entwicklung GmbH, Germany
{timos,siwu,rehm,htor}@informatik.tu-chemnitz.de, htor@cs.indiana.edu, schickhj@de.ibm.com

ABINIT on Cell - Overview

- The Cell/B.E. processor (aka "Cell") developed by Sony, Toshiba and IBM is a heterogenous multicore processor.
- This architecture offers a great peak performance for scientific computations
- We took some opportunities to optimize ABINIT for Cell and present first results

ABINIT:
- A software package to compute the total energy, charge density and electronic structure of systems made of electrons and nuclei
- 240.000 lines of Fortran code
- Uses MPI for parallelization

Project Goals:
1. Run ABINIT on PPE of a single Cell
2. Make good use of the SPEs
3. Run ABINIT on a cluster of Cells
4. Evaluate how ABINIT could benefit from a hybrid multiprocessor architecture

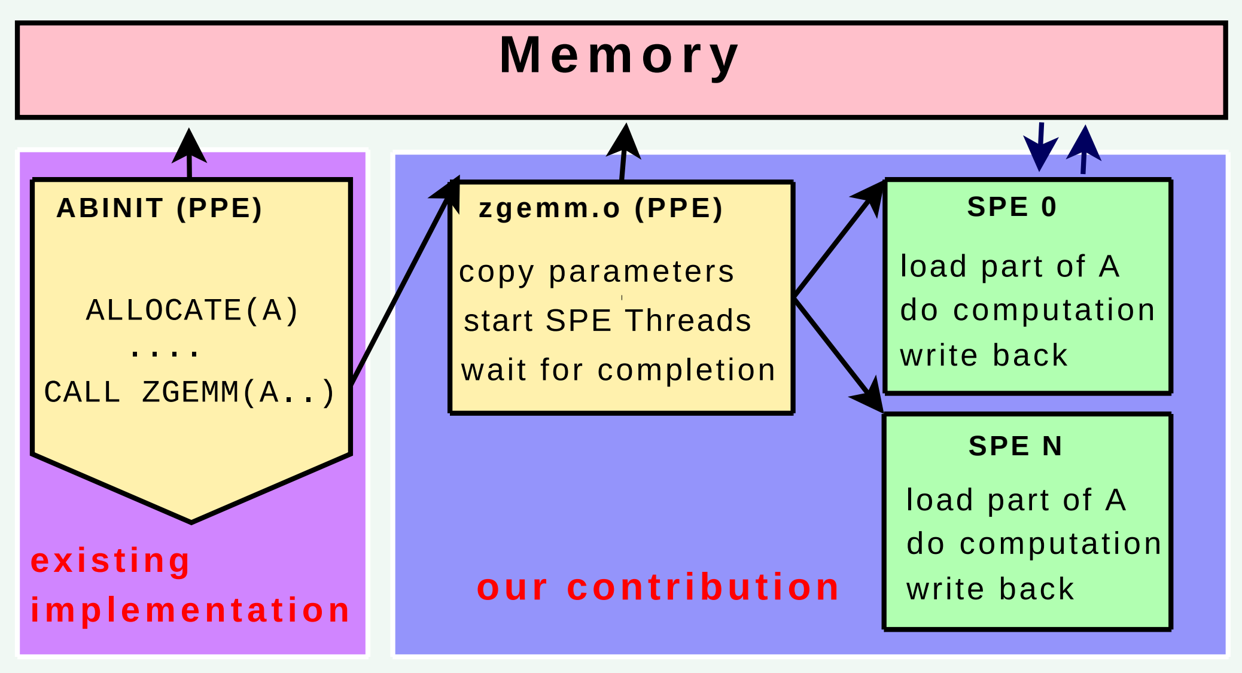
Profiling ABINIT promised that optimizing a few functions should lead to a serious speedup of the whole application, in fact 4765 (2%) source lines of code (SLOC) make up 87% of ABINIT runtime.

Function	Runtime	Task	SLOC
ZGEMM	25%	matrix multiplication	415
opernl4	35%	applying the non local operator	1800
fftstp	15%	fast fourier transformation	1450
mkffkg3	7%	fast fourier transformation	580
pw_orthon	5%	Gram-Schmidt orthogonalization	520

We started by optimizing ZGEMM because the operation which is done by this routine can be understood quite easily, so we could focus on optimization and getting familiar with the Cell programming environment.

Math kernel optimization

Our BLAS3/ZGEMM implementation:
- Parallel multiplication of complex matrices with double precision
- Whole computation is done on the SPEs, PPE only administers SPE threads



- Divide the input matrices into blocks that fit into an SPEs local store
- The actual data partitioning scheme is less significant, the algorithm demands much more multiplications than memory operations
- The innermost loop (where we multiply) must be optimal, so the dual issue rate and number of pipeline stalls are important

$$\begin{aligned}
c_{ij} &= \sum_k a_{ik}\cdot b_{kj} = \sum_k \left(\mathrm{Re}(a_{ik}) + \mathrm{i}\,\mathrm{Arg}(a_{ik})\right)\left(\mathrm{Re}(b_{kj}) + \mathrm{i}\,\mathrm{Arg}(b_{kj})\right)\\
&= \sum_k \mathrm{Re}(a_{ik})\,\mathrm{Re}(b_{kj}) + \mathrm{Re}(a_{ik})\,\mathrm{Arg}(b_{kj})\mathrm{i} + \mathrm{Arg}(a_{ik})\,\mathrm{Re}(b_{kj})\mathrm{i} - \mathrm{Arg}(a_{ik})\,\mathrm{Arg}(b_{kj})\\
&= \sum_k \left[\mathrm{Re}(a_{ik})\,\mathrm{Re}(b_{kj}) - \mathrm{Arg}(a_{ik})\,\mathrm{Arg}(b_{kj})\right] + \left[\mathrm{Re}(a_{ik})\,\mathrm{Arg}(b_{kj}) + \mathrm{Arg}(a_{ik})\,\mathrm{Re}(b_{kj})\right]\mathrm{i}\\
&= \left[\mathrm{Re}(a_{ik})\,\mathrm{Re}(b_{kj}) - \left(\mathrm{Arg}(a_{ik})\,\mathrm{Arg}(b_{kj}) - \mathrm{Re}\left(\sum_{k-1} a_{ik}\cdot b_{kj}\right)\right)\right] +\\
&\quad \left[\mathrm{Arg}(a_{ik})\,\mathrm{Re}(b_{kj}) + \left(\mathrm{Re}(a_{ik})\,\mathrm{Arg}(b_{kj})\right) + \mathrm{Re}\left(\sum_{k-1} a_{ik}\cdot b_{kj}\right)\right]\mathrm{i}
\end{aligned}$$

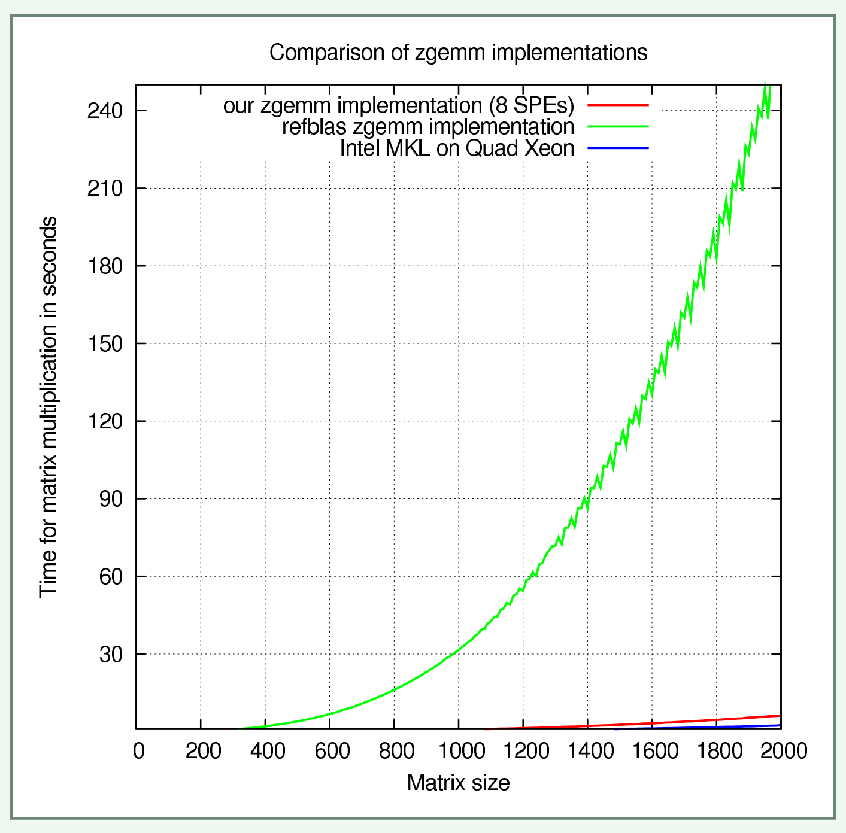
The last equation can be computed with only 4 fused multiply add (FMADD) instructions, compared to 4 multiply and 2 add instructions in line 3.

The complex multiplication described above, implemented in C:

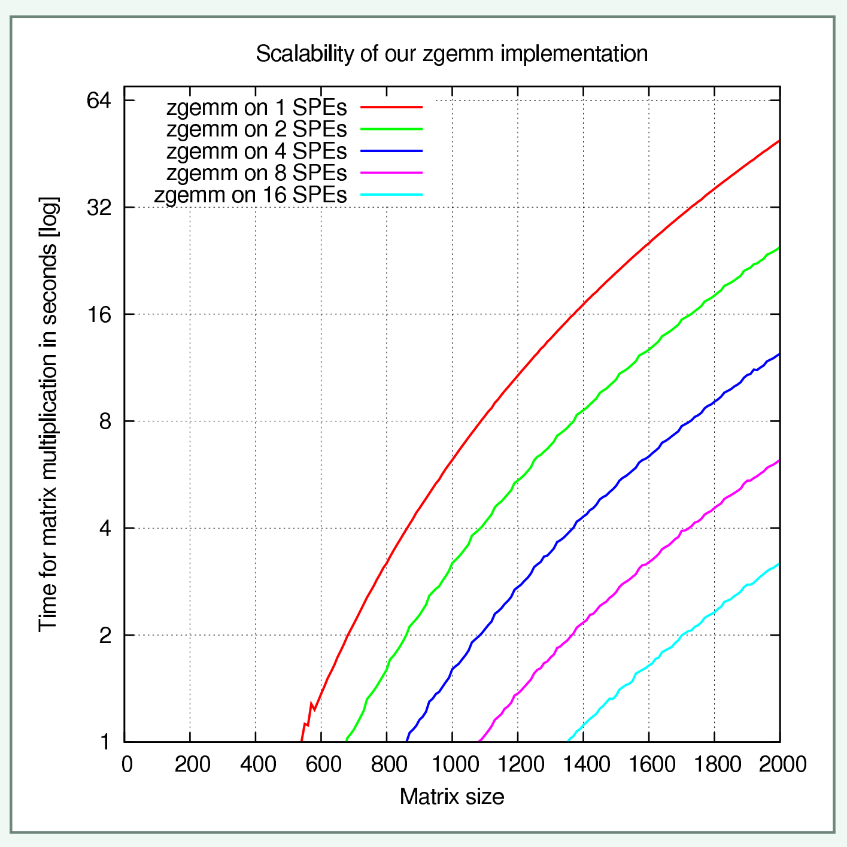
```c
#define VPTR "(vector double *)"
vector char high_double = {0, 1, 2, 3, 4, 5, 6, 7, 16,17,18,19,20,21,22,23};
vector char low_double  = {8, 9,10,11,12,13,14,15, 24,25,26,27,28,29,30,31};
vector double rre, rim, tre, tim;

for (k=0; k < klen; k++, aa += astep, bb += bstep) {
    fim = spu_shuffle( *(VPTR aa), *(VPTR (aa+atstep)), low_double);
    gim = spu_shuffle( *(VPTR bb), *(VPTR bb),          low_double);
    fre = spu_shuffle( *(VPTR aa), *(VPTR (aa+atstep)), high_double);
    gre = spu_shuffle( *(VPTR bb), *(VPTR bb),          high_double);
    tre = spu_msub( fim, gim, rre);
    tim = spu_madd( fre, gim, rim);
    rre = spu_msub( fre, gre, tre);
    rim = spu_madd( fim, gre, tim);
}
tre = spu_shuffle(rre, rim, high_double);
tim = spu_shuffle(rre, rim, low_double);
```

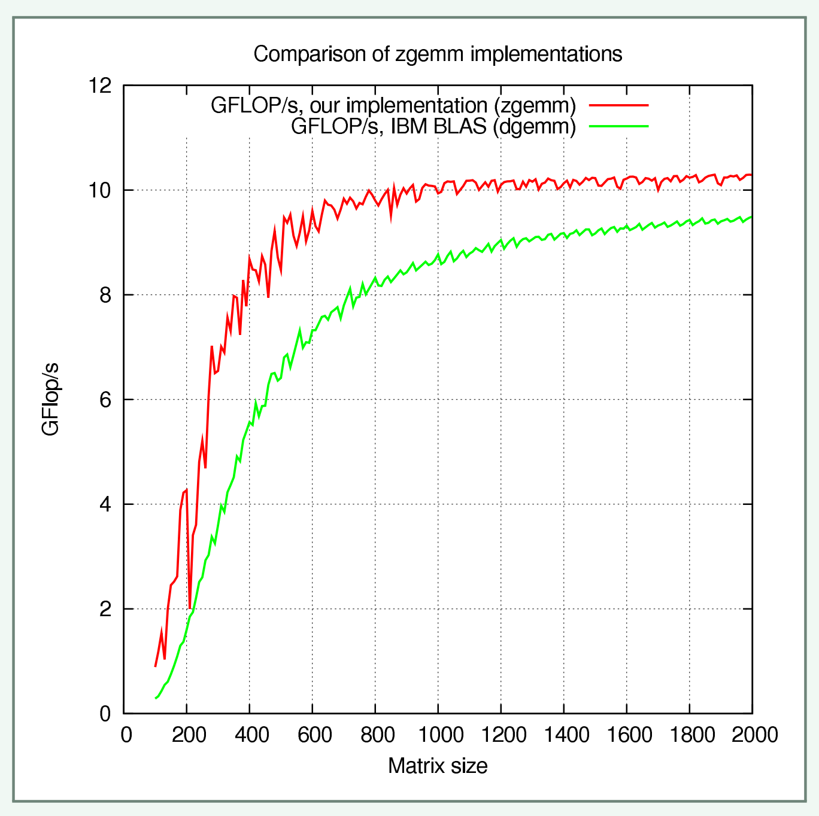
Benchmark results



Our ZGEMM implementation is 40 times faster for 2000x2000 sqare matrices than the ZGEMM implementation in the refblas package



We achieve linear speedup with our ZGEMM implementation, which is due to the good memory/CPU coupling on the Cell architecture



The Cell SDK 3.0 (pre-release) achieves 9.5 GFlop/s DGEMM[a] performance for a 2000x2000 matrix. This corresponds to 68% of the Cell's peak performance. Our optimized ZGEMM implementation is able to leverage up to 73.5% of the peak performance even though the complex multiplication requires more shuffle operations.

[a] The current SDK does not offer a ZGEMM implementation, thus we used DGEMM for comparison.

The unmodified version of ABINIT is roughly twice as fast on a 2 GHz Opteron as on a Cell. If we manage to optimize the other compute kernels by the same factor as ZGEMM the Cell version could be more than three times faster on the Cell than on the Opteron.[b]

To simplify the process of porting math kernels to the Cell plattform we are currently about to build tools which help with optimizing the compiler generated (`gcc -S`) assembly, similar to `spu_timing` but in a more "active" way, which means that the pipeline status should not only be viewable but optimizations should be suggested.

[b] We simulated a test system in a 54^3 FFT box with 108 atoms.

Future Work

- Optimization of the other ABINIT compute kernels for Cell
- Exploring ways to efficiently use heterogenous clusters for ABINIT

This research is supported by the Center for Advanced Studies (CAS) of the IBM Böblingen Laboratory as part of the NICOLL Project.