RapidMind & PGI Accelerator Compiler

Dr. Volker Weinberg
Leibniz-Rechenzentrum der Bayerischen Akademie der Wissenschaften
volker.weinberg@lrz.de

PRACE Workshop “New Languages & Future Technology Prototypes”
LRZ, 1.-2. March 2010
1. The RapidMind Development Platform

2. RapidMind Implementation and Perf. of the 3 EuroBen Kernels
   - mod2am: Dense Matrix-Matrix Multiplication
   - mod2as: Sparse Matrix-Vector Multiplication
   - mod2f: 1-D Complex FFT

3. The PGI Accelerator Compiler

4. PGI Implementation and Perf. of 2 EuroBen Kernels

5. Summary, References and Acknowledgements
The RapidMind Development Platform

- The only platform that allows to write code that runs both on multi- and many-core (Intel + AMD) x86 processors and (ATI + NVIDIA) GPGPUs and the CELL BE → enables to write efficient highly portable code that runs on various architectures.
- Development platform for expressing data-parallel computations (no task parallelism) from within a (single-sourced) ISO standard C++ program.
- SPMD streaming model.
- Integrates with existing C++ compilers, no new tools and compilers are required.
- No need for low-level understanding of the target architecture.
- Code is optimised via dynamic runtime compilation.
- RapidMind Inc. has lately been acquired by Intel and their product will dissolve in Intel’s new language Ct (C for throughput computing).
Basic RapidMind Types: Values & Arrays

Value = Container for fixed-length data

**Value**

1. **half**
2. **double**
3. **float**
4. **int**

**RapidMind Type**

**Tuple Size**

**Element Type**

**Value4f**

= Value **4f** (1.2f, 3.4f, 5.6f, 7.8f);

<table>
<thead>
<tr>
<th></th>
<th>1.2</th>
<th>3.4</th>
<th>5.6</th>
<th>7.8</th>
</tr>
</thead>
</table>

**Array** = Container for RapidMind Values

**Array**

1. **Value4d**
2. **Value3f**
3. **Value2i**

**RapidMind Type**

**Dimensionality**

**Item Type**

**Array**

= **Array**<2, Value3f>(2,4);

<table>
<thead>
<tr>
<th></th>
<th>1.2</th>
<th>3.4</th>
<th>5.6</th>
<th>9.9</th>
<th>2.3</th>
<th>4.7</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2.3</td>
<td>1.4</td>
<td>4.3</td>
<td>5.8</td>
<td>3.1</td>
<td>7.4</td>
</tr>
<tr>
<td></td>
<td>3.8</td>
<td>9.3</td>
<td>2.7</td>
<td>1.2</td>
<td>3.4</td>
<td>5.6</td>
</tr>
<tr>
<td></td>
<td>6.1</td>
<td>4.4</td>
<td>8.3</td>
<td>5.9</td>
<td>3.0</td>
<td>8.6</td>
</tr>
</tbody>
</table>
Array Accessors

- take\((A, 2, 3)\)
- shift\((A, 2, 1)\)
- stride\((A, 2, 1)\)
- offset\((A, 2, 1)\)
- slice\((A, 1, 0, 4, 2)\)

Array\(<2, T>\) \(A(4, 4)\)
```cpp
#include <rapidmind/platform.hpp>
using namespace RapidMind;
...
// declaration
Array<1, Value4i> input;
Array<1, Value4f> output;
Program example = BEGIN {
    // program definition
} END;
// program call
output = example(input);
```
mod2am

Dense Matrix-Matrix Multiplication
mod2am: Simple Implementation

\[ C_{ij} = \sum_k A_{ik} \times B_{kj} \]

Array\langle 2, \text{Value1f} \rangle A(m, l);
Array\langle 2, \text{Value1f} \rangle B(l, n);
Array\langle 2, \text{Value1f} \rangle C(m, n);

Program \text{mxm} = \text{BEGIN} \{
    \text{In}\langle \text{Value2i} \rangle \text{ind};
    \text{Out}\langle \text{Value1f} \rangle c = \text{Value1f}(0.);
    
    \text{Value1i} k;
    // Computation of \text{C}(i,j)
    \text{RM\_FOR} (k = 0, k < \text{Value1i}(1), k++) \{
        c += A[\text{Value2i}(\text{ind}(0), k)] \times B[\text{Value2i}(k, \text{ind}(1))];
    \} \text{RM\_ENDFOR};
\} \text{END};

C = \text{mxm}(\text{grid}(m, n));
2 improved RapidMind implementations

- 2 different versions were needed to achieve acceptable performance
  - GPU-optimised version: part of code that used to be available on the RapidMind developer site rm-sgemm-gpu-5938.zip was used as a reference: uses Value4f values to store matrices, 4x4 submatrices are multiplied and accumulated.
  - CELL-optimised version: code that used to be available on the RapidMind developer site rm-sgemm-cell-5938.zip was used as a reference: Computation is performed using a block partitioning of 64 by 64 blocks. All matrices are in a "block swizzled" format, so that these blocks are contiguous in memory. The computations and memory transfers are overlapped using DoubleBuffering. Based on /opt/cell/sdk/src/demos/matrix_mul/ in the IBM CELL SDK.

- CUDA implementation: uses NVIDIA’s CuBLAS library
- MKL implementation: uses cblas_dgemm
## Hardware Overview

<table>
<thead>
<tr>
<th>Hardware</th>
<th>SP peak perf.</th>
<th>DP peak perf</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nehalem-EP (2.53 GHz, 1 core)</td>
<td>20 GFlop/s</td>
<td>10 GFlop/s</td>
</tr>
<tr>
<td>Nehalem-EP (2.53 GHz, 8 cores)</td>
<td>162 GFlop/s</td>
<td>81 GFlop/s</td>
</tr>
<tr>
<td>1 C1060 GPU</td>
<td>933 GFlop/s</td>
<td>78 GFlop/s</td>
</tr>
<tr>
<td>1 PowerXCell8i (8 SPUs)</td>
<td>205 GFlop/s</td>
<td>102 GFlop/s</td>
</tr>
<tr>
<td>2 PowerXCell8i (16 SPUs)</td>
<td>410 GFlop/s</td>
<td>205 GFlop/s</td>
</tr>
</tbody>
</table>

Volker Weinberg, LRZ

1.3.2010

RapidMind & PGI Accelerator Compiler
Dense matrix-matrix multiplication (mod2am, simple version)

- RM gpu-opt version (sp)
- cuda, 1 C1060, sp
- cuda, 1 C1060, dp
- x86, 8 N-EP, sp
- x86, 8 N-EP, dp

Matrix size (m)

GFlop/s
mod2am: Performance of Improved Implementations

Volker Weinberg, LRZ
LRZ · 1.3.2010
RapidMind & PGI Accelerator Compiler
mod2as
Sparse Matrix-Vector Multiplication
Storage format for input matrix A:
3 array variation of the CSR (compressed sparse row) format:
- `matvals[]`: array that contains the non-zero elements of A
- `indx[i]`: number of the column in A that contains `matvals[i]`
- `rowp[j]`: index of the element in `matvals[]` that is the first nonzero element in row j of A

Implementation:
- **RapidMind implementation**: similar to simple mod2am implementation, uses `Value1f`
- **CUDA implementation**: based on: N. Bell, M. Garland: “Efficient Sparse Matrix-Vector Multiplication on CUDA”
  [www.nvidia.com/object/nvidia_research_pub_001.html](http://www.nvidia.com/object/nvidia_research_pub_001.html)
- **MKL implementation**: uses `cblas_dgemm`
mod2as: RapidMind Implementation

Array<1,Value1i> indx(nelmts);
Array<1,Value1i> rowp(nrows+1);
Array<1,Value1f> matvals(nelmts);
Array<1,Value1f> invvec(ncols);
Array<1,Value1f> outvec(nrows);

Program spMXV = BEGIN {
    In<Value1i> i;
    Out<Value1f> c;

    c = Value1f(0.);
    Value1i j;

    RM_FOR(j=rowp[i], j < rowp[i+1] , j++) {
        c += matvals[j] * invvec[indx[j]];
    } RM_ENDFOR;
} END;

outvec = spMXV(grid(nrows));
mod2as: Performance Comparison

Sparse matrix-vector multiplication (mod2as, dp)

- MKL (6 N-EP cores)
- CUDA (C1060 GPU)
- RapidMind (8 N-EP cores)
- RapidMind (C1060 GPU)
- RapidMind (Cell, 8 SPU)

Volker Weinberg, LRZ
LRZ - 1.3.2010
RapidMind & PGI Accelerator Compiler
mod2f

1-D complex FFT
Implementation of Radix-2 DIF FFT

Discrete Fourier Transformation (DFT):

\[ F(k) = F_N(k, f) = \sum_{n=0}^{N-1} f(n)e^{-2\pi i kn/N} \]

Cooley-Tukey algorithm: FFT technique that recursively breaks down a DFT of size \( N \) into smaller DFTs

Radix-2: divide into 2 FFTs of size \( N/2 \) at each recursion level

Decimation in Frequency: divide into even/odd-numbered frequencies \( k \)

\[ F_N(k, f) = \begin{cases} & F_{N/2}(k/2, f_e) \\ & F_{N/2}((k-1)/2, f_o) \end{cases} \]

for \( k \) even

for \( k \) odd

with

\[ f_e(n) = f(n) + f(n + N/2) \]

\[ f_o(n) = (f(n) - f(n + N/2))e^{-2\pi in/N} \]

FFT Butterfly Kernel
Implementation of Radix-2 DIF FFT

\( N = 2^n = 8 \)

Simple Implementation

\( f(0) \rightarrow f_0(0) \rightarrow f_{0e}(0) \rightarrow f_{e0}(0) \rightarrow f_{ee}(0) \rightarrow F(0) \)
\( f(1) \rightarrow f_0(1) \rightarrow f_{0e}(1) \rightarrow f_{e0}(1) \rightarrow f_{ee}(0) = F(4) \)
\( f(2) \rightarrow f_0(2) \rightarrow f_{0e}(0) \rightarrow f_{e0}(0) \rightarrow f_{ee}(0) = F(2) \)
\( f(3) \rightarrow f_0(3) \rightarrow f_{0e}(1) \rightarrow f_{e0}(1) \rightarrow f_{ee}(1) \rightarrow F(3) \)
\( f(4) \rightarrow f_0(0) \rightarrow f_{0e}(1) \rightarrow f_{e0}(0) \rightarrow f_{ee}(0) = F(6) \)
\( f(5) \rightarrow f_0(1) \rightarrow f_{0e}(0) \rightarrow f_{e0}(0) \rightarrow f_{ee}(1) \rightarrow F(5) \)
\( f(6) \rightarrow f_0(2) \rightarrow f_{0e}(0) \rightarrow f_{e0}(0) \rightarrow f_{ee}(1) \rightarrow F(7) \)
\( f(7) \rightarrow f_0(3) \rightarrow f_{0e}(0) \rightarrow f_{e0}(0) \rightarrow f_{ee}(1) \rightarrow F(7) \)

Volker Weinberg, LRZ

RapidMind & PGI Accelerator Compiler
Implementation of Radix-2 DIF FFT

\[ N = 2^n = 8 \]

Reordering of FFT butterfly operations:

\[ f(0) \quad f(0) \quad f(0) \quad f(0) \quad f(0) \quad f(0) \quad f(0) \quad f(0) \]

\[ f(1) \quad f(4) \quad e^{2\pi i \frac{1}{8}} f(4) \quad e^{2\pi i \frac{1}{8}} f(1) \quad e^{2\pi i \frac{1}{8}} f(2) \quad e^{2\pi i \frac{1}{8}} f(3) \quad e^{2\pi i \frac{1}{8}} f(5) \quad e^{2\pi i \frac{1}{8}} f(6) \]

\[ f(2) \quad f(1) \quad f(2) \quad e^{2\pi i \frac{1}{8}} f(2) \quad e^{2\pi i \frac{1}{8}} f(2) \quad e^{2\pi i \frac{1}{8}} f(3) \quad e^{2\pi i \frac{1}{8}} f(5) \quad e^{2\pi i \frac{1}{8}} f(6) \]

\[ f(3) \quad f(5) \quad f(3) \quad f(5) \quad f(3) \quad f(5) \quad f(3) \quad f(5) \]

\[ f(6) \quad f(3) \quad f(6) \quad f(3) \quad f(6) \quad f(3) \quad f(6) \quad f(3) \]

\[ f(7) \quad f(7) \quad f(7) \quad f(7) \quad f(7) \quad f(7) \quad f(7) \quad f(7) \]

input stream interleaving  input stream interleaving  input stream interleaving  tangling

\[ f_{out}(0) = F(0) \quad f_{out}(0) = F(1) \quad f_{out}(0) = F(2) \quad f_{out}(0) = F(3) \]

\[ f_{out}(0) = F(4) \quad f_{out}(0) = F(5) \quad f_{out}(0) = F(6) \quad f_{out}(0) = F(7) \]
$N = 2^n = 8$

Reordering back-to-front, split streams
mod2f: RapidMind Implementation

rapidmind::set_default_boundary_mode(BOUNDARY_REPEAT);

Array<1, Value2f> data(N); // N=2^n dim input data

// Helper array caches
vector<Array<1, Value2f>> twiddles; // n-dim vector
// twiddles[i] contains 2^i twiddle factors
vector<Array<1, Value1ui>> bitreverses; // (n+1)-dim vector,
// contains mapping info for bit reversing

Array<1, Value2f> src;
src = tangle(bitreverses[n]);

for (int k=n-1; k>=0; k--) {
   Array<1, Value2f> dest(N);
   // write into lower half of output array
   take(dest,N/2) = Butterfly1(stride(src, 2), stride(offset(src, 1), 2));
   // write into upper half of output array
   offset(dest,N/2) = Butterfly2(stride(src, 2), stride(offset(src, 1), 2),
                                 take(twiddles[k], N/2));
   // swap src and dest buffer
   src=dest;
}

Volker Weinberg, LRZ
LRZ - 1.3.2010
RapidMind & PGI Accelerator Compiler
Tangling of input data

```c
    tangle = BEGIN {
        In<Value1ui> index;
        Out<Value2f> c;
        c = data[index];
    } END;
```

FFT Butterfly1: Addition

```c
    Butterfly1 = BEGIN {
        In<Value2f> a, b;
        Out<Value2f> c = a + b;
    } END;
```

FFT Butterfly2: Substraction & Multiplication with Twiddle Factors

```c
    Butterfly2 = BEGIN {
        In<Value2f> a, b, w;
        Value2f t = a - b;
        Out<Value2f> c;
        c[0] = t[0]*w[0] + t[1]*w[1];
        c[1] = t[1]*w[0] - t[0]*w[1];
    } END;
```
Implementation:

- **RapidMind implementation**: Split-Stream Radix-2 DIF (Decimation in Frequency) FFT, uses Cooley-Tukey algorithm, uses Value2f to store complex values, based on code optimised by RapidMind Inc.,

- **CUDA implementation**: uses NVIDIA’s CuFFT library

- **MKL implementation**: uses DftiComputeForward
mod2f: Performance Comparison

Fast Fourier Transformation (mod2f, sp)

- MKL (1 N-EP core)
- CUDA (C1060 GPU)
- RapidMind (C1060 GPU)
- RapidMind (8 N-EP cores)
- RapidMind (Cell, 8 SPU's)

Length vs. GFlop/s
The PGI Accelerator Compiler

PGI Accelerator Programming Model:

- Does for GPU programming what OpenMP did for POSIX threads,
- high-level implicit model for x64+ NVIDIA CUDA GPU systems,
- supported both for pgcc (C99) and pgf95 (Fortran 95) compilers,
- uses directives (C pragmas, Fortran comments) to offload compute-intensive code to an accelerator,
- building programs:
  - in C: `pgcc -o c1 c1.c -ta=nvidia -Minfo=accel -Msafeptr -fast`
  - in F90: `pgfortran -o f1 f1.f90 -ta=nvidia -Minfo=accel -fast`
PGI Accelerator Directives

- **Accelerator Compute Region Directive**
  Defines the region of the program that should be compiled for execution on the accelerator device.

  ```
  C
  #pragma acc region [clause [, clause] ...] !$acc region [clause [, clause] ...] {
  ...
  } $acc end region
  
  Fortran
  !$acc region [clause [, clause] ...] ...
  $acc end region
  ```

  Important clauses:
  - **copyin(list)**: Variables, arrays or subarrays in the list need to be copied to GPU memory.
  - **copyout(list)**: Variables, arrays or subarrays in the list need to be copied back to the host memory.
  - **copy(list)**: Combination of copyin and copyout.
  - **local(list)**: Local variables/arrays in GPU memory not needed on the host.
PGI Accelerator Directives

**Accelerator Loop Mapping Directive**
Describes what type of parallelism to use to execute the loop on the following line.

C

```c
#pragma acc for [clause [, clause] ...] 
{ 
  for(i=0; i<n; i++){ ...
    ...
  }
}
```

Fortran

```fortran
$acc do [clause [, clause] ...]
  do i=1, n
    ...
  enddo
$acc end do
```

**important clauses:**

- `parallel[(width)]`: Execute this loop in parallel mode (MIMD parallelisation across multiprocessors) on the accelerator.
- `vector[(width)]`: Execute this loop in vector mode (SIMD vectorization within a multiprocessor, threads within a thread block) on the accelerator.
- `seq[(width)]`: Execute this loop sequentially on the accelerator.
- `host[(width)]`: Execute the loop sequentially on the host processor.
void mxm( int lda, int m, int l, int n, double a[lda], double b[n], double c[n] ) {
    int i, j, k;
    #pragma acc region
    for( j = 0; j < n; j++ ) {
        for( i = 0; i < m; i++ ) {
            c[i][j] = 0.0;
            for( k = 0; k < n; k++ ) {
                c[i][j] = c[i][j] + a[i][k]*b[k][j];
            }
        }
    }
}

pgcc -ta=nvidia:cc13 -Minfo=accel -fast -Msafeptr=all -O3 -c -o mxm.o mxm.c

mxm:
3, Generating copyin(b[0:n-1][0:n-1])
   Generating copyin(a[0:m-1][0:n-1])
   Generating copyout(c[0:m-1][0:n-1])
4, Loop is parallelizable
   Accelerator kernel generated
   4, #pragma acc for parallel, vector(256)
5, Loop is parallelizable
7, Complex loop carried dependence of 'c' prevents parallelization
   Loop carried reuse of 'c' prevents parallelization
   Inner sequential loop scheduled on accelerator

weinberg@lx64tv1:~/LRZ2/mod2am-pgi> export ACC_NOTIFY=1; ./x.mod2am
launch kernel file=/home/weinberg/LRZ2/mod2am-pgi/mxm.c function=mxm line=4 device=0
grid=20 block=256

5000 | 5000 | 5000| 21.73 | 1.15e+04 | T |
```c
1 void mxm( int lda, int m, int l, int n, double a[][lda], double b[][n], double c[][n] ) {
2     int i, j, k;
3     #pragma acc region
4     for( j = 0; j < n; j++ ) {
5         for( i = 0; i < m; i++ ) {
6             c[i][j] = 0.0;
7             for( k = 0; k < n; k++ ) {
8                 c[i][j] = c[i][j] + a[i][k]*b[k][j];
9             }
10         }
11     }
12 }
```

```
pgcc -ta=nvidia:cc13 -Minfo=accel -fast -Msafeptr=all -O3 -c -o mxm.o mxm.c
mxm:

3, Generating copyin(b[0:n-1][0:n-1])
   Generating copyin(a[0:m-1][0:n-1])
   Generating copyout(c[0:m-1][0:n-1])
4, Loop is parallelizable
   Accelerator kernel generated
   4, #pragma acc for parallel, vector(256)
5, Loop is parallelizable
7, Complex loop carried dependence of 'c' prevents parallelization
   Loop carried reuse of 'c' prevents parallelization
   Inner sequential loop scheduled on accelerator
```

```
weinberg@lx64tv1:~/LRZ2/mod2am-pgi> export ACC_NOTIFY=1; ./x.mod2am
launch kernel  file=/home/weinberg/LRZ2/mod2am-pgi/mxm.c function=mxm line=4 device=0
grid=20 block=256

5000 | 5000 | 5000| 21.73 | 1.15e+04 | T |
```

Volker Weinberg, LRZ
void spmxv( int nrows, int nelmts, int indx[], int rowp[], double matvals[],
    double invvec[], double outvec[] ) {

    int i, j, ncols=nrows;
    #pragma acc region for parallel,vector(256) copyout( outvec[0:nrows-1] ),
        copyin(matvals[0:nelmts-1], indx[0:nelmts-1], invvec[0:ncols-1], rowp[0:nrows] )
    for( i = 0; i < nrows ; i++ ){
        outvec[i]=0.;
        for( j = rowp[i]; j < rowp[i+1]; j++ ){
        }
    }
}

pgcc -ta=nvidia:cc13 -Minfo=accel -fast -Msafeptr=all -O3 -c -ospmxv.o spmxv.c
  Generating copyin(rowp[:nrows])
  Generating copyout(outvec[:nrows-1])
  Generating copyin(matvals[:nelmts-1])
  Generating copyin(invec[:ncols-1])
  Generating copyin(indx[:nelmts-1])

4, Loop is parallelizable
   Accelerator kernel generated
   #pragma acc for parallel, vector(256)
      Cached references to size [257] block of 'rowp'
      Using register for 'outvec'

6, Complex loop carried dependence of 'outvec' prevents parallelization
   Loop carried reuse of 'outvec' prevents parallelization
   Inner sequential loop scheduled on accelerator

weinberg@lx64tv1:~/LRZ2/mod2as-pgi> ./x.mod2as
launch kernel  file=/home/weinberg/LRZ2/mod2as-pgi/spmxv.c function=spmxv line=4 device=0
grid=16 block=256
void spmxv( int nrows, int nelmts, int indx[], int rowp[], double matvals[],
    double invec[], double outvec[] ) {

    int i, j, ncols=nrows;
    #pragma acc region for parallel,vector(256) copyout( outvec[0:nrows-1] ),
      copyin(matvals[0:nelmts-1], indx[0:nelmts-1], invec[0:ncols-1], rowp[0:nrows] )
    for( i = 0; i < nrows ; i++ ){
      outvec[i]=0.;
      for( j = rowp[i]; j < rowp[i+1]; j++ ){
      }
    }
}

pgcc -ta=nvidia:cc13 -Minfo=accel -fast -Msafeptr=all -O3 -c -ospmxv.o spmxv.c
  3, Generating copyin(rowp[0:nrows])
    Generating copyout(outvec[0:nrows-1])
    Generating copyin(matvals[0:nelmts-1])
    Generating copyin(invec[0:ncols-1])
    Generating copyin(indx[0:nelmts-1])
  4, Loop is parallelizable
    Accelerator kernel generated
      #pragma acc for parallel, vector(256)
      Cached references to size [257] block of 'rowp'
      Using register for 'outvec'
  6, Complex loop carried dependence of 'outvec' prevents parallelization
    Loop carried reuse of 'outvec' prevents parallelization
    Inner sequential loop scheduled on accelerator

weinberg@lx64tv1:~/LRZ2/mod2as-pgi> ./x.mod2as
launch kernel  file=/home/weinberg/LRZ2/mod2as-pgi/spmxv.c function=spmxv line=4 device=0
  grid=16 block=256
Sparse matrix-vector multiplication (mod2as)

Performance in GFlop/s

#Rows

- CUDA, dp
- RapidMind, sp
- RapidMind, cp
- PGI, sp
- PGI, dp
RapidMind is the only platform that allows to write code that runs both on multi-core CPUs and GPGPU/CELL B.E.. RapidMind is easy to pick up and only few lines of code are needed to express an algorithm, however, the performance among platforms is very different.

Different implementations were necessary to reach acceptable performance on the different platforms.

RapidMind has lately been acquired by Intel and their product will dissolve in Intel’s new language Ct.

PGI Accelerator compiler does for GPU programming what OpenMP did for POSIX threads; uses directives (C pragmas, Fortran comments) to offload compute-intensive code to an NVIDIA CUDA GPU system.
M. McCool, K. Wadleigh, B. Henderson, H.-Y. Lin
*Performance evaluation of GPUs using the RapidMind development platform*
Proceedings of the 2006 ACM/IEEE conf. on Supercomputing, 2006

N. Bell, M. Garland
*Efficient Sparse Matrix-Vector Multiplication on CUDA*
www.nvidia.com/object/nvidia_research_pub_001.html

T. Jansen, B. von Rymon-Lipinski, N. Hanssen, E. Keeve
*Fourier Volume Rendering on the GPU Using a Split-Stream-FFT*
VMV 2004: 395-403

Iris Christadler, Volker Weinberg
*RapidMind: Portability across Architectures and its Limitations*
Acknowledgements

- Matthias Brehm, **Iris Christadler**, Hans Hacker (CUDA, MKL port), LRZ, Germany
- KONWIHR-II project OMI4papps: Optimisation, Modelling and Implementation for Highly Parallel Applications
- PRACE project funded in part by the EU’s 7th Framework Programme (FP7/2007-2013) under grant agreement no. RI-211528