A brief introduction to HONEI

Danny van Dyk, Markus Geveler, Dominik Göddeke,
Carsten Gutwenger, Sven Mallach, Dirk Ribbrock

March 2009

Contents

1 Introduction

This tutorial addresses programmers, who want to use HONEI's existing operations to build their own applications as well as those, who want to implement and add their own application specific kernels. Consequently, the tutorial is roughly divided into two parts. Sections 2 and 3.1 address the former developers, providing a hands-on introduction as well as information on how to compose kernels from existing ones. Advanced topics such as adding new hardware-specific implementations to HONEI are covered in Sections 3.2, 3.3 and 3.4.

It should be noted, that this tutorial explains using and extending HONEI and not primarily *optimisations* of a specific operation, like loop unrolling etc.

2 Using HONEI

This section addresses users who simply want to build their applications on top of HONEI's operations. It provides a brief overview of how to set up and access containers and how to use the components of the libraries. It is supposed that HONEI has been installed and set up properly as explained in the README file located in the HONEI tarball.

In order to make use of HONEI's highly tuned linear algebra, math or application libraries, the user needs to know some things about its containers and operation interfaces. The first example demonstrates, how to set up some containers to store floating-point data for later usage. Suppose a vector norm of a generalised residual is needed, aka $s = ||\alpha \mathbf{y} + \beta A \mathbf{x}||$. For this first example two different types of containers are used, namely a vector type and, for simplicity, a (generic) banded-matrix type with three bands: The diagonal and the first super- and subdiagonals. The sample code can be found in Listing 1.

After including the operation header files all data is stored in vectors of type `DenseVector` as the banded matrix type is designed to contain these as non-zero bands. All containers are tagged with their data-type and, in this example, are invoked using the constructor that only takes the size of the vector (note, that the matrix takes the size of the diagonal band, which is the same as for the vector) as arguments. The band-vectors are then added to the matrix by using `BandedMatrix`' `insert_band()` method, which uses an intuitive numbering of its bands. The vectors can now be accessed in exactly the same way as arrays and HONEIs operations can be applied to it. Here, the SSE backend is used for this example. Note that it is not necessary to deal with data alignment, because HONEI takes care of it. It is assumed here, that the data is about to be used again elsewhere in the application and due to this, the vectors and the matrix shall be invariant under the operations. Most current HONEI operations work in situ and consequently at first a copy of $\mathbf{y}$ has to be made using the copy-constructor in combination with `DenseVector`'s `copy()` method to avoid $\mathbf{y}$ being overwritten. The copy then serves as the result vector that is to be reduced. The operations are used by calling their static `value()` methods and the hardware tag specifies the specialisation to use. Some operations are called using more template parameters, for example, the `Norm`, where the programmer has to specify the norm type and its mode.

Listing 1: Using HONEI - A first application example

```
#include<honei/la/norm.hh>
#include<honei/la/scale.hh>
#include<honei/la/scaled_sum.hh>
#include<honei/la/product.hh>

using namespace honei;
.
.
//your application code
.
.
//setting up the data:
double alpha(0.5), beta(0.2), s;

unsigned long size(1000);
DenseVector<double> y(size);
DenseVector<double> x(size);

DenseVector<double> A_diagonal(size);
DenseVector<double> A_upper(size);
DenseVector<double> A_lower(size);

//Fill the vectors:
for(unsigned long i(0) ; i < size ; ++i)
{
    y[i] = ...
    x[i] = ...
    A_diagonal[i] = ...
    A_upper[i] = ...
    A_lower[i] = ...
}

BandedMatrix<double> A(size);
A.insert_band(0, A_diagonal);
A.insert_band(1, A_upper);
A.insert_band(-1, A_lower);

//begin computation:
DenseVector<double> result(y.copy());

result = Scale<tags::CPU::SSE>::value(result, alpha);
result = ScaledSum<tags::CPU::SSE>
        ::value(result, Product<tags::CPU::SSE>::value(A, x), beta);

s = Norm<vnt_l_two, false, tags::CPU::SSE>::value(result);
.
.
.
```

3 Developing HONEI kernels

Now, that the very basics of using HONEI have been described, the same operation, namely computing the norm described in section 2 is stepwisely added to the HONEI math library `libhoneimath`.

3.1 Writing a composite operation

At first, the series of HONEI operation calls can be encapsulated in order to provide a new operation, `ScaledProductSumNorm`, with the abovementioned property of being a read-only operation in the sense, that no container's data is altered. In addition, a nine banded tridiagonal matrix is used in contrast to the very simple example of the last section.The sample code is presented in Listing 2 and should be deployed in a header file with suffix `.hh` within the folder `honei/math/`. Finally, the corresponding line in the `honei/math/files.m4` file has to be added to make HONEI's build system recognise it. The template containing the value method and the method itself each have one template parameter: The class template is defined with the type of the backend and the value method with the data-type parameter. The latter's arguments are straightforward except the newly introduced `BandedMatrixQ1` type instance. Note that the computational part is exactly the same as in the first example despite returning s.

Although writing composite operations is quite useful (take into account, that the operation is written once and provided for all backends), the most important feature of an application specific kernel is *increased performance*. For sparse linear algebra, the most important way to achieve this is by fusing operations and taking hardware features into account, i.e., by avoiding to stream the data through the memory hierarchy several times. The demonstration of the process of adding such a kernel is continued for each backend separately.

Listing 2: Developing kernels - A composite kernel

```
#include<honei/la/norm.hh>
#include<honei/la/scale.hh>
#include<honei/la/scaled_sum.hh>
#include<honei/la/product.hh>
#include <honei/util/memory_arbiter.hh>
#include <honei/la/algorithm.hh>
#include <honei/util/tags.hh>

namespace honei
{
    template<typename Tag_>
        struct ScaledProductSumNorm
        {
            template<typename DataType_>
                static inline DataType_ value(DataType_ alpha,
                                              DenseVector<DataType_> & y,
                                              DataType_ beta,
                                              BandedMatrixQ1<DataType_> & A,
                                              DenseVector<DataType_> & x)
                {
                    DenseVector<DataType_> result(y.copy());
                    result = Scale<Tag_>::value(result, alpha);
                    result = ScaledSum<Tag_>
                        ::value(result, Product<Tag_>::value(A, x), beta);

                    return Norm<vnt_l_two, false, Tag_>::value(result);

                }
        };
}
```

3.2 Adding application specific kernels: SSE

In order to add a new kernel implementation to the SSE backend, the programmer has to accomplish the following:

- A template specialisation has to be added to the header file, which contains a declaration of the `value()` methods only. For the example kernel, the code in Listing 3 achieves this. `value()` methods for both double and single precision have to be provided, because their implementation will be slightly different.

- A new file `scaled_product_sum_norm-sse.cc` has to be added to the same folder. This file contains the implementation of the value methods, except the part, that has to be redirected to HONEI's SSE backend, see Listing 4 for the float version. In this tutorial and for the sake of simplicity, a partly optimised version of the example routine is presented (For a fully optimised version, see the code provided with the tarball): The matrix-vector product is computed by calling HONEI's corresponding product. Then, the (aligned) data contained by the resulting vector (and **y** respectively) are assigned to a correctly typed pointer by the `elements()` method and passed to a new method in the `honei::sse` namespace. The only new code consists of calls to the containers' `lock()` and `unlock()` methods. This is necessary because, in principle, the new operation can be called using every backend. In particular, that means that it is possible, that more than one backend, especially more than one *device* takes part in the surrounding application's computations. In order to guarantee data consistency, HONEI's memory arbiter has to know, that a device is about to read from the containers. Note that HONEI takes care of all necessary transfers from and to the device (e.g. the GPU) by itself. The only thing the programmer has to take care of is locking data appropriately when necessary (besides the need for knowing the total amount of memory the device actually has). It should be noted, that without specifying the memory target, the default is CPU main memory. The data is invariant under this operation, hence the lock mode is `lm_read_only`. The `sse` statement in the corresponding `files.m4` can now be added.

- The signature of the new backend operation has to be added to the `operations.hh` file in the `honei/backends/sse` folder.

- So far, the effort has been limited to minimal changes. A new file `scaled_product_sum_norm.cc` has to be added to the `honei/backends/sse` folder. It contains the implementations of the methods. The sample code in Listing 5 displays the single precision version. Note, that the effort *in this stage* is usually much higher.

- At last, the new file has to be added to the `Makefile.am` file within the backend directory.

Listing 3: Developing kernels - Adding the specialisation for SSE

```
namespace honei
{
    .
    .
    .

    template<>
        struct ScaledProductSumNorm_TUTORIAL<tags::CPU::SSE>
        {
            static float value(float a,
                               DenseVector<float> & y,
                               float b,
                               BandedMatrixQ1<float> & A,
                               DenseVector<float> & x);

            static double value(double a,
                                DenseVector<double> & y,
                                double b,
                                BandedMatrixQ1<double> & A,
                                DenseVector<double> & x);
        };
    .
    .
    .

}
```

Listing 4: Developing kernels - Redirection to the SSE backend

```
#include <honei/backends/sse/operations.hh>
#include <honei/math/scaled_product_sum_norm.hh>
#include <honei/la/product.hh>

namespace honei
{
    float ScaledProductSumNorm_TUTORIAL<tags::CPU::SSE>
        ::value(float a,
                DenseVector<float> & y,
                float b,
                BandedMatrixQ1<float> & A,
                DenseVector<float> & x)
    {

        //Still use HONEIs BandedMatrix-DenseVector product:
        DenseVector<float> A_x(Product<tags::CPU::SSE>::value(A, x));

        x.lock(lm_read_only);
        y.lock(lm_read_only);
        A.lock(lm_read_only);

        //do not care about alignment, the used HONEI containers
        //provide aligned data
        float * A_x_data = A_x.elements();
        float * y_data = y.elements();

        //redirect the relevant data to the SSE backend
        float result(honei::sse::scaled_product_sum_norm(x.size(),
                                                         a,
                                                         y_data,
                                                         b,
                                                         A_x_data));

        x.unlock(lm_read_only);
        y.unlock(lm_read_only);
        A.unlock(lm_read_only);

        return result;

    }
}
```

Listing 5: Developing kernels - SSE Implementation

```
#include <honei/util/attributes.hh>

#include <xmmintrin.h>
#include <emmintrin.h>

namespace honei {

    namespace sse {

        float scaled_product_sum_norm(unsigned long size, float a,
                                      float * y,
                                      float b,
                                      float * A_x)
        {

            union sse4
            {
                __m128 m;
                float f[4];
            } m1, m2, m3, m4, m5;

            float result(0);

            //compute ||ay + bA_x|| (SSE)
            m1.m = _mm_set_ps1(a);
            m2.m = _mm_set_ps1(b);

            m5.m = _mm_setzero_ps();

            unsigned long quad_end(size - (size % 4));
            for (unsigned long i(0) ; i < quad_end ; i += 4)
            {
                m3.m = _mm_load_ps(A_x + i);
                m4.m = _mm_load_ps(y + i);

                m4.m = _mm_mul_ps(m1.m, m4.m);
                m3.m = _mm_mul_ps(m2.m, m3.m);

                m3.m = _mm_add_ps(m3.m, m4.m);
                m3.m = _mm_mul_ps(m3.m, m3.m);

                m5.m = _mm_add_ps(m5.m, m3.m);
            }

            result += m5.f[0];
            result += m5.f[1];
            result += m5.f[2];
            result += m5.f[3];

            //compute ||ay + bA_x|| (FPU)
            for (unsigned long i(quad_end) ; i < size ; ++i)
            {
                result += (a * y[i] + b * A_x[i]) * (a * y[i] + b * A_x[i]);
            }

            return result;
        }
    }
}
```

3.3 Adding application specific kernels: GPU

The steps in the previous subsection are exactly the same as those needed for adding a kernel to the CUDA backend, except that the file containing the implementation has suffix `.cu`. No additional efforts concerning the buildsystem have to be made. Again, HONEI's matrix-vector product is used and a backend function in `honei/backends/cuda` is called, that processes the norm on the GPU. It is abstained from presenting all listings, but concentrated on the implementational part in Listing 6. Now that a device with off-chip memory is used, HONEI's memory arbiter returns the device memory address of the locked container data. Again, the data is then passed to a new method. HONEI's `Configuration` class can be used to determine the needed values for block- and gridsize, see the README file for details on that. The CUDA specific code is presented in Listings 7 and 8. It is recommended, that the `CUDA_ERROR()` macro is used, which processes CUDA runtime errors and throws HONEI exceptions.

Listing 6: Developing kernels - CUDA header

```
#include <honei/math/scaled_product_sum_norm.hh>
#include <honei/backends/cuda/operations.hh>
#include <honei/util/memory_arbiter.hh>
#include <honei/util/configuration.hh>


using namespace honei;

float ScaledProductSumNorm_TUTORIAL<tags::GPU::CUDA>
        ::value(float a,
                DenseVector<float> & y,
                float b,
                BandedMatrixQ1<float> & A,
                DenseVector<float> & x)
{
        //Still use HONEIs BandedMatrix-DenseVector product:
        DenseVector<float> A_x(Product<tags::GPU::CUDA>::value(A, x));

        unsigned long blocksize(Configuration::instance()
                ->get_value("cuda::spsn_float", 128ul));
        unsigned long gridsize(Configuration::instance()
                ->get_value("cuda::spsn_float_grid", 16ul));
        void * A_x_gpu (A_x.lock(lm_read_only,
                                 tags::GPU::CUDA::memory_value));
        void * y_gpu (y.lock(lm_read_only,
                                 tags::GPU::CUDA::memory_value));

        //redirect the relevant data to the CUDA backend
        float result(cuda_scaled_product_sum_norm_float_tut(x.size(),
                                                            a,
                                                            y_gpu,
                                                            b,
                                                            A_x_gpu,
                                                            blocksize,
                                                            gridsize));

        y.unlock(lm_read_only);
        A_x.unlock(lm_read_only);

        return result;
}
```

Listing 7: Developing kernels - CUDA Implementation

```
#include <honei/backends/cuda/cuda_util.hh>

namespace honei
{
    namespace cuda
    {
        __global__ void
                scaled_product_sum_norm_tut_gpu(float a,
                                                float * y,
                                                float b,
                                                float * A_x,
                                                float * tmp,
                                                unsigned long size,
                                                unsigned long blocksize)
        {
            // calculate how many elements each thread needs to calculate
            const unsigned long iter = size / (blockDim.x * gridDim.x);
            unsigned long pos = blockIdx.x* blocksize + threadIdx.x;

            // clear the output
            tmp[blockIdx.x * blocksize + threadIdx.x] = 0;

            for (unsigned long i = 0 ; i < iter ; ++i)
            {
                tmp[blockIdx.x * blocksize + threadIdx.x]
                    += (A_x[pos] * b + y[pos] * a) *
                        (A_x[pos] * b + y[pos] * a);
                pos += blockDim.x * gridDim.x;
            }

            // for the last iteration, check if the elements are still
                available
            if (pos < size)
            {
                tmp[blockIdx.x * blocksize + threadIdx.x]
                    += (A_x[pos] * b + y[pos] * a) *
                        (A_x[pos] * b + y[pos] * a);
            }
        }
    }
}
```

Listing 8: Developing kernels - CUDA Implementation (continued)

```
extern "C" float
        cuda_scaled_product_sum_norm_float_tut(unsigned long size,
                                               float a,
                                               void * y,
                                               float b,
                                               void * A_x,
                                               unsigned long blocksize,
                                               unsigned long gridsize)
{
    float result(0.);

    dim3 grid(gridsize);
    dim3 block(blocksize);
    float * A_x_gpu((float* )A_x);
    float * y_gpu((float* )y);
    float * tmp_cpu(0);
    float * tmp_gpu(0);

    cudaMalloc((void**)&tmp_gpu, gridsize * blocksize * sizeof(float));
    cudaMallocHost((void**)&tmp_cpu, gridsize * blocksize * sizeof(float));

    honei::cuda
        ::scaled_product_sum_norm_tut_gpu<<<grid, block>>>(a,
                                                           y_gpu,
                                                           b,
                                                           A_x_gpu,
                                                           tmp_gpu,
                                                           size,
                                                           blocksize);

    cudaMemcpy(tmp_cpu,
               tmp_gpu,
               blocksize * gridsize * sizeof(float),
               cudaMemcpyDeviceToHost);

    for (unsigned long i(0) ; i < blocksize * gridsize ; ++i)
    {
        result += tmp_cpu[i];
    }

    cudaFree(tmp_gpu);
    cudaFreeHost(tmp_cpu);

    CUDA_ERROR();
    return result;
}
```

3.4 Adding application specific kernels: Cell

To add a kernel to the Cell backend, the programmer in principle has to do the same as in the above two subsections (despite the fact, that the directory containing the files is `honei/backends/cell/spe/math/`). In addtition, two particularities of HONEI's Cell backend have to be dealt with:

- The operation code (`opcode`) has to be added to the `honei/backends/opcodes.hh` file. It starts with a '`oc_`' that is followed by the operation's name, so in case of the example operation, it is '`oc_scaled_product_sum_norm_float`'.
- In `honei/backends/cell/spe/kernels/kernel_float.sk`, the operation has to be associated with a kernel explicitly since the operations are scattered among several kernels to lower the Local Store's usage. Adding the line '`dword scaled_product_sum_norm`' to the libmath section in the `.sk` file achieves this.

A closer look at the file containing the SPE code for this operation, `scaled_product_sum_norm_float.cc`, displayed in Listing 9 completes this tutorial. Again, the kernel deals with computing the linear combination and its vector norm after receiving the result of the matrix-vector product. Multiply-accumulate SIMD operations in combination with simple multiplications are used in the code that has to be added to the `honei::cell::implementation` namespace and hence it is as self-explanatory as other SIMD code (e.g. in the SSE example). Furthermore, HONEI again deals with transfers from and to the SPEs. This is achieved by using the `Operation` class template. An instance of this template is generated in the `honei::cell::operations` namespace and the number of non-scalar operands (the two vectors), the data type and the transfer type are specified. In the case of the example reduction only a scalar has to be transfered back to main memory. Hence, it can be mailed back and no DMA transfer is needed to receive the result.

Listing 9: Developing kernels - CELL Implementation

```
#include <honei/backends/cell/cell.hh>
#include <honei/backends/cell/spe/libla/operations.hh>
#include <honei/backends/cell/spe/libutil/allocator.hh>
#include <honei/backends/cell/spe/libutil/transfer.hh>

#include <spu_intrinsics.h>

namespace honei
{
    namespace cell
    {
        namespace implementation
        {
            vector float
               scaled_product_sum_norm_float(const vector float &
                   accumulator,
                                             vector float & b_carry,
                                             const vector float * a,
                                             const vector float * b,
                                             unsigned size,
                                             unsigned b_offset,
                                             float alpha,
                                             float beta)
            {
                vector float acc = accumulator;
                const vector float alphav = spu_splats(alpha);
                const vector float betav = spu_splats(beta);

                for (unsigned i(0) ; i < size ; ++i)
                {
                    vector float b_with_offset = b_carry;
                    extract(b_with_offset, b[i], b_offset);

                    vector float sum
                       = spu_madd(alphav, a[i], spu_mul(betav,
                           b_with_offset));
                    acc = spu_madd(sum, sum, acc);

                    b_carry = b[i];
                }

                return acc;
            }
        }

        namespace operations
        {
            Operation<2, float, rtm_mail> scaled_product_sum_norm_float = {
                &zero_float,
                &implementation::scaled_product_sum_norm_float,
                &sum_float
            };
        }
    }
}
```