Kernels in gemmlowp

Kernels provide an inner-loop implementation, and a format

Here we assume familiarity with the concepts of kernels and of packing as explained in design.md.

gemmlowp is designed to be easily extensible to different architectures and other low-level details, while achieving high performance. Thus a line had to be drawn between the generic GEMM code and the specific parts that need to be manually designed for each architecture, etc. The design choice made in gemmlowp is to have easily swappable GEMM kernels.

In itself, a GEMM kernel is just an implementation of the inner-most loop in a GEMM (That inner-most loop has to be over the 'depth' dimension so as to be able to accumulate into a small enough number of accumulators to fit in registers).

Thus, by itself, a GEMM kernel should be just a function computing a block of GEMM.

However, GEMM kernels may need to differ not just in how they implement this computation, but also in the format of data that they operate on. Indeed, in order to maximize the ratio of arithmetic instructions to memory access instructions, GEMM kernels want to handle blocks as wide as possible given the number of registers of the CPU architecture.

Thus, in order to allow efficient specialization to diverse architectures, gemmlowp allows each GEMM kernel to dictate the format of data that it expects, in addition to providing its inner-loop implementation.

The former is given by a 'Format' typedef, and the latter by a 'Run' method.

A good example is to look at internal/kernel_neon.h, and specifically at the NEONKernel12x4Depth2 kernel, which specifies its format as

```
typedef KernelFormat<KernelSideFormat<CellFormat<4, 2>, 3>,
                     KernelSideFormat<CellFormat<4, 2>, 1> > Format;
```

The meaning of these terms is explained in the lengthy comment at the top of internal/kernel.h. Here, they mean that this kernel handles at each iteration (along the depth dimension): - 3 'cells' of size 4x2 each of the lhs, so a total lhs block of size 12x2 - 1 'cell' of size 2x4 of the rhs. In other words, this kernel handles 12 rows of the lhs and 4 columns of the rhs, and handles two levels of depth at once. The 'cells' and `CellFormat` detail the layout of these 12x2 and 2x4 blocks.

This kernel then loads these 12x2 and 2x4 blocks and computes the corresponding 12x4 GEMM; for ease of reference let us paste the critical comment and code here:

```
"loop_NEONKernel12x4Depth2_%=:\n"

// Overview of register layout:
//
// A 2x4 cell of Rhs is stored in 16bit in d0--d1 (q0).
// A 12x2 block of 3 4x2 cells Lhs is stored in 16bit in d2--d7
// (q1--q3).
// A 12x4 block of accumulators is stored in 32bit in q4--q15.
//
//                  +-----+-----+-----+-----+
```

```
//                   |d0[0]|d0[1]|d0[2]|d0[3]|
//              Rhs  +-----+-----+-----+-----+
//                   |d1[0]|d1[1]|d1[2]|d1[3]|
//                   +-----+-----+-----+-----+
//
//                   |     |     |     |     |
//
//    Lhs            |     |     |     |     |
//
//  +--+--+ - - - -  +-----+-----+-----+-----+
//  |d2|d3|          | q4  | q5  | q6  | q7  |
//  |d2|d3|          | q4  | q5  | q6  | q7  |
//  |d2|d3|          | q4  | q5  | q6  | q7  |
//  |d2|d3|          | q4  | q5  | q6  | q7  |
//  +--+--+ - - - -  +-----+-----+-----+-----+
//  |d4|d5|          | q8  | q9  | q10 | q11 |
//  |d4|d5|          | q8  | q9  | q10 | q11 |
//  |d4|d5|          | q8  | q9  | q10 | q11 |
//  |d4|d5|          | q8  | q9  | q10 | q11 |
//  +--+--+ - - - -  +-----+-----+-----+-----+
//  |d6|d7|          | q12 | q13 | q14 | q15 |
//  |d6|d7|          | q12 | q13 | q14 | q15 |
//  |d6|d7|          | q12 | q13 | q14 | q15 |
//  |d6|d7|          | q12 | q13 | q14 | q15 |
//  +--+--+ - - - -  +-----+-----+-----+-----+
//
//                            Accumulator

// Load 1 Rhs cell of size 2x4
"vld1.8 {d0}, [%[rhs_ptr]:64]!\n"

// Load 3 Lhs cells of size 4x2 each
"vld1.8 {d2}, [%[lhs_ptr]:64]!\n"
"vld1.8 {d4}, [%[lhs_ptr]:64]!\n"
"vld1.8 {d6}, [%[lhs_ptr]:64]!\n"

// Expand Lhs/Rhs cells to 16 bit.
"vmovl.u8 q0, d0\n"
"vmovl.u8 q1, d2\n"
"vmovl.u8 q2, d4\n"
"vmovl.u8 q3, d6\n"

// Multiply-accumulate, level of depth 0
"vmlal.u16 q4, d2, d0[0]\n"
"vmlal.u16 q5, d2, d0[1]\n"
"vmlal.u16 q6, d2, d0[2]\n"
"vmlal.u16 q7, d2, d0[3]\n"
"vmlal.u16 q8, d4, d0[0]\n"
"vmlal.u16 q9, d4, d0[1]\n"
"vmlal.u16 q10, d4, d0[2]\n"
"vmlal.u16 q11, d4, d0[3]\n"
"vmlal.u16 q12, d6, d0[0]\n"
"vmlal.u16 q13, d6, d0[1]\n"
"vmlal.u16 q14, d6, d0[2]\n"
"vmlal.u16 q15, d6, d0[3]\n"

// Multiply-accumulate, level of depth 1
"vmlal.u16 q4, d3, d1[0]\n"
"vmlal.u16 q5, d3, d1[1]\n"
"vmlal.u16 q6, d3, d1[2]\n"
"vmlal.u16 q7, d3, d1[3]\n"
"vmlal.u16 q8, d5, d1[0]\n"
"vmlal.u16 q9, d5, d1[1]\n"
"vmlal.u16 q10, d5, d1[2]\n"
"vmlal.u16 q11, d5, d1[3]\n"
"vmlal.u16 q12, d7, d1[0]\n"
"vmlal.u16 q13, d7, d1[1]\n"
"vmlal.u16 q14, d7, d1[2]\n"
"vmlal.u16 q15, d7, d1[3]\n"

// Loop. Decrement loop index (depth) by 2, since we just handled 2
// levels of depth (Kernel::kDepth=2).
"subs %[run_depth], #2\n"
"bne loop_NEONKernel12x4Depth2_%=\n"
```

Packing code adapts to the format chosen by the kernel

As explained in design.md, gemmlowp starts by packing blocks of the lhs and rhs matrices for optimally efficient traversal by the kernel. This depends on fine details of the kernel format, in ways that can only be efficiently handled by knowing these kernel format details at compile-time.

This is the reason why all the code in internal/pack.h is templated in the corresponding kernel format.

The code in internal/pack.h isn't tightly optimized by itself, but it is structured in such a way that the critical code is in a template, `PackingRegisterBlock`, that can easily be specialized to override the slow generic code with fast specific packing code for specific formats, on specific platforms.

See internal/pack_neon.h which provides NEON specializations of the packing code for the particular kernel formats that are used by the NEON kernels in internal/kernel_neon.h.

Wrapping up: how to optimize gemmlowp for a CPU architecture

In conclusion, the key feature of gemmlowp when it comes to efficiently supporting a specific CPU architecture, is that it allows to freely replace the inner loop of the GEMM by providing one's own GEMM kernel, which is also free to dictate its required data layout; each data layout then also needs optimized packing code. The steps are thus:

1. Freely design a GEMM kernel with a freely chosen data layout.
2. Implement the GEMM kernel, similar to internal/kernel_neon.h.
3. Implement the optimized packing code, similar to internal/pack_neon.h.