Overview of gemmlowp design

Primer on GEMM, kernels, and cache friendliness

gemmlowp, like most GEMMs, implements the straightforward matrix multiplication algorithm, which takes n^3 multiply-accumulate instructions for n*n sized matrices. Because the arithmetic complexity grows quicker than the memory complexity (n^3 vs. n^2), memory accesses are redundant (each matrix entry is accessed n times). A large part of a GEMM's performance and design goes toward minimizing the inefficiency resulting from these redundant memory accesses.

Ultimately, once values are loaded into CPU registers, they cost nothing to access, so as long as we can work within registers, this problem doesn't exist. Thus, in order to be efficient, a GEMM's inner loops must wisely use the available registers to do as much arithmetic work as possible before loading more data from memory into registers. This means that a GEMM implementation needs to have architecture-specific inner loops tailored for architecture details such as the number of registers, and typically written in assembly. This 'inner loops' architecture-specific component is referred to as the GEMM kernel. (More details about kernels are in kernel.md).

However, only small blocks can fit at a given time in registers, so at larger scales one needs to repeatedly load blocks of matrices from memory, and these accesses are redundant for the reason outlined above. The way that one minimizes the resulting inefficiency is by organizing for cache locality, so that most of these accesses hit the L1 cache, and most of the remaining ones hit the L2 cache, etc.

This is achieved by subdividing the matrices into blocks sized to fit in L2 cache, and subdividing these blocks into sub-blocks sizes to fit in L1 cache, and performing the matrix multiplication one such block at a time.

In practice, it tends to pay off to "pack" input blocks for optimally efficient traversal by the kernel, since they will be traversed multiple times. "packing" means at least reordering the data layout for 1) simple access patterns that fit the CPU's cache behavior (in particular, the cache line size), and 2) simple loading into SIMD vector registers by the kernel.

So a typical GEMM, in pseudo-code, tends to look like this:

```
allocate(some_lhs_L2_block);
allocate(some_rhs_L2_block);
for (some_lhs_L2_block) {
  pack(some_lhs_L2_block);
  for (some_rhs_L2_block) {
    pack(some_rhs_L2_block);
    for (some_lhs_sub_block in some_lhs_L2_block) {
      for (some_rhs_sub_block in some_rhs_L2_block) {
        kernel(some_lhs_sub_block, some_rhs_sub_block);
      }
    }
  }
}
```

Impact of low-precision computation on gemmlowp design

Refer to low-precision.md for specifics of the low-precision-computation paradigm and how it's implemented in gemmlowp.

Inputs and outputs are matrices of uint8 values, but internally we are accumulating int32 values, only converting them back to uint8 at the end. This means that we need so store a block of int32 accumulators at a time. We compute a block of the result in int32 accumulators and then we "unpack" it into the destination matrix at once. In this way, we minimize the amount of memory used to store int32 values at a given time.

Because of that, besides the "pack" and "kernel" stages outlined above, a third stage is needed in gemmlowp, which we call "unpack". Thus we arrive at the 3-stage computation scheme that gemmlowp uses:

1. Pack lhs/rhs blocks from the input matrices.
2. Compute the product of the packed blocks, using the kernel.
3. Unpack the result block into the output matrix.

The pseudo-code overview of gemmlowp now looks like:

```
allocate(some_lhs_L2_block);
allocate(some_rhs_L2_block);
// new: temp storage for int32 accums
allocate(some_int32_accumulators_block);
for (some_lhs_L2_block) {
  pack(some_lhs_L2_block);
  for (some_rhs_L2_block) {
    pack(some_rhs_L2_block);
    for (some_lhs_sub_block in some_lhs_L2_block) {
      for (some_rhs_sub_block in some_rhs_L2_block) {
        // new: pass int32 accums to kernel
        kernel(&some_int32_accumulators_block,
               some_lhs_sub_block,
               some_rhs_sub_block);
      }
    }
    // new: unpack int32 accums into destination matrix
    unpack(some_int32_accumulators_block);
  }
}
```

Exploring gemmlowp code

The design outlined above can be readily matched to gemmlowp source code, in particular in this file, which gives a simple GEMM implementation fitting in one rather small function:

```
internal/single_thread_gemm.h
```

The reader can compare the above pseudo-code to the actual code in this file:

```
for (int r = 0; r < rows; r += block_params.l2_rows) {
  int rs = std::min(block_params.l2_rows, rows - r);

  PackLhs(&packed_lhs, lhs.block(r, 0, rs, depth));

  for (int c = 0; c < cols; c += block_params.l2_cols) {
    int cs = std::min(block_params.l2_cols, cols - c);

    if (!pack_rhs_once) {
      PackRhs(&packed_rhs, rhs.block(0, c, depth, cs));
    }

    Compute(kernel, block_params, &packed_result, packed_lhs, packed_rhs);

    auto result_block = result->block(r, c, rs, cs);
    UnpackResult(&result_block, packed_result, packed_lhs, packed_rhs, depth,
                 result_offset, result_mult_int, result_shift);
  }
}
```

The files in `internal/` fall into a few categories:

There are two top-level GEMM implementations,

- internal/single_thread_gemm.h
- internal/multi_thread_gemm.h

They both call into pack/compute/unpack stages (see kernel.md and packing.md) implemented in the following files:

- internal/pack.h
- internal/compute.h
- internal/unpack.h
    - This in turn calls into internal/output.h for the output pipeline (see output.md)

The pack.h and unpack.h files contain generic templated code that can be overridden by optimized code in template specializations; for example, see the NEON optimized code here:

- internal/pack_neon.h
- internal/unpack_neon.h
    - This in turn calls into internal/output_neon.h

The compute stage contains generic code in compute.h that only calls into optimized code through the Kernel::Run() entry point. Each kernel is basically just as struct offering a Run() implementation; see the NEON kernels in:

- internal/kernel_neon.h