```
1  // Copyright 2016 The Gemmlowp Authors. All Rights Reserved.
2  //
3  // Licensed under the Apache License, Version 2.0 (the "License");
4  // you may not use this file except in compliance with the License.
5  // You may obtain a copy of the License at
6  //
7  //     http://www.apache.org/licenses/LICENSE-2.0
8  //
9  // Unless required by applicable law or agreed to in writing, software
10 // distributed under the License is distributed on an "AS IS" BASIS,
11 // WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
12 // See the License for the specific language governing permissions and
13 // limitations under the License.
14
15 // This is a standalone testbed and benchmark for gemmlowp-style GEMM kernels,
16 // either doing integer or float arithmetic.
17 //
18 // It verifies that a kernel produces correct results, then benchmarks it;
```

```
19  // if multiple CPU cores are detected, it tries to benchmark on each core.
20  // This is aimed at big.LITTLE systems.
21  //
22  // This program is entirely self-contained, and can be compiled manually
23  // such as suggested in the command lines below.
24  // It currently supports only Android/ARM but would trivially generalize to
25  // other OSes (it's mostly standard POSIX) or architectures (each kernel
26  // targets a specific architecture, one may simply add more).
27
28  /*
29  Build and run this benchmark on Android/ARM/32bit:
30  export CXX=~/android/toolchains/arm-linux-androideabi/bin/arm-linux-androideabi-clang++
31  $CXX -fPIE -static -O3 --std=c++11 neon-gemm-kernel-benchmark.cc -o benchmark -mfloat-abi=softfp -mfpu=neon-vfpv4 && adb push be
32
33  Build and run this benchmark on Android/ARM/64bit:
34  export CXX=~/android/toolchains/aarch64-linux-android/bin/aarch64-linux-android-clang++
35  $CXX -fPIE -static -O3 --std=c++11 neon-gemm-kernel-benchmark.cc -o benchmark && adb push benchmark /data/local/tmp && adb shell
36  */
37
38  #include <sched.h>
39  #include <unistd.h>
40
41  #include <type_traits>
42  #include <random>
43  #include <algorithm>
44  #include <cstdint>
45  #include <iostream>
46  #include <cstdlib>
47  #include <cassert>
48
49  #ifdef PRINT_CPUFREQ
50  #include <fstream>
51  #include <sstream>
52  #include <string>
53  #endif
54
```

```
55  #if !defined __arm__ && !defined __aarch64__
56  #error This benchmark assumes ARM (for inline assembly sections).
57  #endif
58
59  // Typically one wants to fit in L1 cache, and GEMM implementations
60  // are carefully optimized to tune their access patterns to that effect.
61  // Most devices have at least 16k of L1 cache. The Kraits have exactly 16k.
62  const int kDefaultCacheSizeK = 16;
63
64  const int kCacheLineSize = 64;
65
66  // BEGIN code copied from gemmlowp/internal/kernel.h
67
68  // Explanation of general gemmlowp terminology
69  // ===========================================
70  //
71  // We use the following abbreviations:
72  // LHS = "left-hand side"
73  // RHS = "right-hand side"
74  // Sometimes when referring to either LHS or RHS, we just say a "Side".
75  //
76  // In a matrix product of a MxK matrix times a KxN matrix,
77  // we call K the 'depth'. Note that M is the number of rows
78  // of the result (and of the LHS), and N is the number of columns
79  // of the result (and of the RHS).
80  //
81  // In each of the LHS and RHS matrices, we call 'width' the
82  // other dimension, besides the depth. So in the LHS, 'width'
83  // is the number of rows, while in the RHS, 'width' is the number
84  // of columns.
85  //
86  //  So in the LHS MxK matrix, the depth is K and the width in M.
87  // And in the RHS KxN matrix, the depth is K and the width in N.
88  //
89  // This is illustrated in this picture:
90  //
```

```
91  //                            RHS width
92  //                        <----------------->
93  //                        +-----------------+ ^
94  //                        |       RHS       | | Depth
95  //                        +-----------------+ v
96  //                ^ +--+ +-----------------+
97  //                | |L | |                 |
98  //       LHS width | |H | |      Result     |
99  //                 | |S | |                 |
100 //                 v +--+ +-----------------+
101 //                   <-->
102 //                   Depth
103
104 // Explanation of gemmlowp kernel formats and "cells"
105 // ================================================
106 //
107 // Kernels operate on small LHS and RHS blocks that fit in registers.
108 // These blocks are stored contiguously in memory, but not always
109 // in a traditional column-major or row-major order; instead,
110 // they consist of a number of sub-blocks, which we call "cells",
111 // that are stored in column-major or row-major order. However,
112 // what really matters to us is not so much rows vs columns, but
113 // rather width vs depth. So we refer to "width-major" and "depth-major"
114 // storage orders. In the LHS, width-major means row-major,
115 // while in the RHS, width-major means column-major.
116 // There is also a third possibility, "diagonal order",
117 // which is unused at the moment.
118 //
119 // We aim to treat both sides, LHS and RHS, on an equal footing,
120 // so we call them both 'sides'. A KernelFormat thus is just a pair
121 // of KernelSideFormat's, one for LHS and one for RHS; each KernelSideFormat
122 // contains a CellFormat and a number of cells; cells are only ever
123 // stacked in the width dimension, which means stacked vertically in the
124 // LHS and stacked horizondally in the RHS.
125 //
126 // Example
```

```
127  // =======
128  //
129  // Let's work out the data layout expected by a kernel having the
130  // following format (the struct names here are defined below in this file):
131  //
132  // KernelFormat<
133  //   KernelSideFormat<CellFormat<3, 4>, 3>,
134  //   KernelSideFormat<CellFormat<5, 4>, 2>
135  // >
136  //
137  // The LHS format, KernelSideFormat<CellFormat<3, 4>, 3>, means:
138  // 3 cells, each cell having dimensions (width=3, depth=4), laid out in
139  // DepthMajor order (the default value, see CellFormat). In the LHS,
140  // DepthMajor means column-major, so the LHS cells are of size 3x4 in
141  // column-major order, so the LHS layout is:
142  //
143  // 0  3  6  9
144  // 1  4  7  10
145  // 2  5  8  11
146  // 12 15 18 21
147  // 13 16 19 22
148  // 14 17 20 23
149  // 24 27 30 33
150  // 25 28 31 34
151  // 26 29 32 35
152  //
153  // The RHS format, KernelSideFormat<CellFormat<5, 4>, 2>, means:
154  // 2 cells each having dimensions (width=5, depth=4), laid out in
155  // DepthMajor order (the default value, see CellFormat). In the RHS,
156  // DepthMajor means row-major, so the RHS cells are of size 4x5 in
157  // row-major order, so the RHS layout is:
158  //
159  // 0  1  2  3  4  20 21 22 23 24
160  // 5  6  7  8  9  25 26 27 28 29
161  // 10 11 12 13 14 30 31 32 33 34
162  // 15 16 17 18 19 35 36 37 38 39
```

```
163
164 // CellOrder enumerates the possible storage orders (=layouts) for
165 // a cell (see explanation above).
166 enum class CellOrder { DepthMajor, WidthMajor, Diagonal };
167
168 // CellFormat describes how data is laid
169 // out in a cell. That is, a CellOrder together with actual dimensions.
170 template <int tWidth, int tDepth, CellOrder tOrder>
171 struct CellFormat {
172   static const int kWidth = tWidth;
173   static const int kDepth = tDepth;
174   static const CellOrder kOrder = tOrder;
175
176   static const int kSize = kWidth * kDepth;
177 };
178
179 // KernelSideFormat describes how data is laid out in a kernel side
180 // (i.e. LHS or RHS). That is, a CellFormat together with a number of
181 // cells. These cells are always stacked in the Width dimension.
182 // For example, in the LHS case, the Width dimension is the rows dimension,
183 // se we're saying that in the LHS, cells are stacked vertically.
184 // We never stack cells in the Depth dimension.
185 template <typename tCellFormat, int tCells>
186 struct KernelSideFormat {
187   typedef tCellFormat Cell;
188   static const int kCells = tCells;
189   static const int kWidth = kCells * Cell::kWidth;
190   static const int kDepth = Cell::kDepth;
191 };
192
193 // KernelFormat describes fully the input data layout that a kernel expects.
194 // It consists of two KernelSideFormat's, one for LHS and one for RHS.
195 template <typename tLhs, typename tRhs>
196 struct KernelFormat {
197   typedef tLhs Lhs;
198   typedef tRhs Rhs;
```

```
199
200   static_assert(Lhs::Cell::kDepth == Rhs::Cell::kDepth, "");
201   static const int kDepth = Lhs::Cell::kDepth;
202   static const int kRows = Lhs::Cell::kWidth * Lhs::kCells;
203   static const int kCols = Rhs::Cell::kWidth * Rhs::kCells;
204 };
205
206 inline const char* CellOrderName(CellOrder o) {
207   switch (o) {
208     case CellOrder::DepthMajor:
209       return "DepthMajor";
210     case CellOrder::WidthMajor:
211       return "WidthMajor";
212     case CellOrder::Diagonal:
213       return "Diagonal";
214     default:
215       assert(false);
216       return nullptr;
217   }
218 }
219
220 // Returns the offset into a cell, at which a given coefficient is stored.
221 template <typename CellFormat>
222 inline int OffsetIntoCell(int w, int d) {
223   switch (CellFormat::kOrder) {
224     case CellOrder::DepthMajor:
225       return w + d * CellFormat::kWidth;
226     case CellOrder::WidthMajor:
227       return d + w * CellFormat::kDepth;
228     case CellOrder::Diagonal:
229       assert(CellFormat::kWidth == CellFormat::kDepth);
230       static const int size = CellFormat::kWidth;
231       return ((size + w - d) * size + d) % (size * size);
232     default:
233       assert(false);
234       return 0;
```

```
235     }
236   }
237
238   // END code copied from gemmlowp/internal/kernel.h
239
240   #ifdef __arm__
241
242   // This is the current standard kernel in gemmlowp, see:
243   // https://github.com/google/gemmlowp/blob/b1e2a29ff866680028f3080efc244e10e8dd7f46/internal/kernel_neon.h#L33
244   struct NEON_32bit_GEMM_Uint8Operands_Uint32Accumulators {
245     typedef std::uint8_t OperandType;
246     typedef std::uint32_t AccumulatorType;
247     typedef KernelFormat<KernelSideFormat<CellFormat<4, 2, CellOrder::DepthMajor>, 3>,
248                          KernelSideFormat<CellFormat<4, 2, CellOrder::DepthMajor>, 1> >
249         Format;
250     static void Run(const OperandType* lhs_ptr, const OperandType* rhs_ptr, AccumulatorType* accum_ptr, int depth) {
251       asm volatile(
252         // Load accumulators
253         "mov r0, %[accum_ptr]\n"
254         "vld1.32 {d8, d9},   [r0]!\n"
255         "vld1.32 {d16, d17}, [r0]!\n"
256         "vld1.32 {d24, d25}, [r0]!\n"
257         "vld1.32 {d10, d11}, [r0]!\n"
258         "vld1.32 {d18, d19}, [r0]!\n"
259         "vld1.32 {d26, d27}, [r0]!\n"
260         "vld1.32 {d12, d13}, [r0]!\n"
261         "vld1.32 {d20, d21}, [r0]!\n"
262         "vld1.32 {d28, d29}, [r0]!\n"
263         "vld1.32 {d14, d15}, [r0]!\n"
264         "vld1.32 {d22, d23}, [r0]!\n"
265         "vld1.32 {d30, d31}, [r0]!\n"
266
267         "loop_%=:\n"
268         // Overview of register layout:
269         //
270         // A 2x4 cell of Rhs is stored in 16bit in d0--d1 (q0).
```

```
271         // A 12x2 block of 3 4x2 cells Lhs is stored in 16bit in d2--d7
272         // (q1--q3).
273         // A 12x4 block of accumulators is stored in 32bit in q4--q15.
274         //
275         //                  +-----+-----+-----+-----+
276         //                  |d0[0]|d0[1]|d0[2]|d0[3]|
277         //             Rhs  +-----+-----+-----+-----+
278         //                  |d1[0]|d1[1]|d1[2]|d1[3]|
279         //                  +-----+-----+-----+-----+
280         //
281         //                  |     |     |     |     |
282         //
283         //    Lhs           |     |     |     |     |
284         //
285         //  +--+--+ - - - -  +-----+-----+-----+-----+
286         //  |d2|d3|          | q4  | q5  | q6  | q7  |
287         //  |d2|d3|          | q4  | q5  | q6  | q7  |
288         //  |d2|d3|          | q4  | q5  | q6  | q7  |
289         //  |d2|d3|          | q4  | q5  | q6  | q7  |
290         //  +--+--+ - - - -  +-----+-----+-----+-----+
291         //  |d4|d5|          | q8  | q9  | q10 | q11 |
292         //  |d4|d5|          | q8  | q9  | q10 | q11 |
293         //  |d4|d5|          | q8  | q9  | q10 | q11 |
294         //  |d4|d5|          | q8  | q9  | q10 | q11 |
295         //  +--+--+ - - - -  +-----+-----+-----+-----+
296         //  |d6|d7|          | q12 | q13 | q14 | q15 |
297         //  |d6|d7|          | q12 | q13 | q14 | q15 |
298         //  |d6|d7|          | q12 | q13 | q14 | q15 |
299         //  |d6|d7|          | q12 | q13 | q14 | q15 |
300         //  +--+--+ - - - -  +-----+-----+-----+-----+
301         //
302         //                            Accumulator
303
304         // Load 1 Rhs cell of size 2x4
305         "vld1.8 {d0}, [%[rhs_ptr]]!\n"
306
```

```
307         // Load 3 Lhs cells of size 4x2 each
308         "vld1.8 {d2}, [%[lhs_ptr]]!\n"
309         "vld1.8 {d4}, [%[lhs_ptr]]!\n"
310         "vld1.8 {d6}, [%[lhs_ptr]]!\n"
311
312         // Expand Lhs/Rhs cells to 16 bit.
313         "vmovl.u8 q0, d0\n"
314         "vmovl.u8 q1, d2\n"
315         "vmovl.u8 q2, d4\n"
316         "vmovl.u8 q3, d6\n"
317
318         // Multiply-accumulate, level of depth 0
319         "vmlal.u16 q4, d2, d0[0]\n"
320         "vmlal.u16 q5, d2, d0[1]\n"
321         "vmlal.u16 q6, d2, d0[2]\n"
322         "vmlal.u16 q7, d2, d0[3]\n"
323         "vmlal.u16 q8, d4, d0[0]\n"
324         "vmlal.u16 q9, d4, d0[1]\n"
325         "vmlal.u16 q10, d4, d0[2]\n"
326         "vmlal.u16 q11, d4, d0[3]\n"
327         "vmlal.u16 q12, d6, d0[0]\n"
328         "vmlal.u16 q13, d6, d0[1]\n"
329         "vmlal.u16 q14, d6, d0[2]\n"
330         "vmlal.u16 q15, d6, d0[3]\n"
331
332         // Multiply-accumulate, level of depth 1
333         "vmlal.u16 q4, d3, d1[0]\n"
334         "vmlal.u16 q5, d3, d1[1]\n"
335         "vmlal.u16 q6, d3, d1[2]\n"
336         "vmlal.u16 q7, d3, d1[3]\n"
337         "vmlal.u16 q8, d5, d1[0]\n"
338         "vmlal.u16 q9, d5, d1[1]\n"
339         "vmlal.u16 q10, d5, d1[2]\n"
340         "vmlal.u16 q11, d5, d1[3]\n"
341         "vmlal.u16 q12, d7, d1[0]\n"
342         "vmlal.u16 q13, d7, d1[1]\n"
```

```
343         "vmlal.u16 q14, d7, d1[2]\n"
344         "vmlal.u16 q15, d7, d1[3]\n"
345
346         // Loop. Decrement loop index (depth) by 2, since we just handled 2
347         // levels of depth.
348         "subs %[depth], #2\n"
349         "bne loop_%=\n"
350
351         // Store accumulators
352         "mov r0, %[accum_ptr]\n"
353         "vst1.32 {d8, d9},   [r0]!\n"
354         "vst1.32 {d16, d17}, [r0]!\n"
355         "vst1.32 {d24, d25}, [r0]!\n"
356         "vst1.32 {d10, d11}, [r0]!\n"
357         "vst1.32 {d18, d19}, [r0]!\n"
358         "vst1.32 {d26, d27}, [r0]!\n"
359         "vst1.32 {d12, d13}, [r0]!\n"
360         "vst1.32 {d20, d21}, [r0]!\n"
361         "vst1.32 {d28, d29}, [r0]!\n"
362         "vst1.32 {d14, d15}, [r0]!\n"
363         "vst1.32 {d22, d23}, [r0]!\n"
364         "vst1.32 {d30, d31}, [r0]!\n"
365         :  // outputs
366         [lhs_ptr] "+r"(lhs_ptr), [rhs_ptr] "+r"(rhs_ptr),
367         [depth] "+r"(depth)
368         :  // inputs
369         [accum_ptr] "r"(accum_ptr)
370         :  // clobbers
371         "cc", "memory", "r0",
372         "d0", "d1", "d2", "d3", "d4", "d5", "d6", "d7", "d8", "d9", "d10",
373         "d11", "d12", "d13", "d14", "d15", "d16", "d17", "d18", "d19", "d20",
374         "d21", "d22", "d23", "d24", "d25", "d26", "d27", "d28", "d29", "d30",
375         "d31");
376     }
377 };
378
```

```
379 // This is Maciek Chociej's fast kernel not expanding operands,
380 // from gemmlowp/meta/. Search for
381 //      mul_3x8_3x8_int32_lhsadd_rhsadd
382 // in this file:
383 // https://raw.githubusercontent.com/google/gemmlowp/e4b9d858b6637d5d0058bfa3d869d2b95864251b/meta/single_thread_gemm.h
384 struct NEON_32bit_GEMM_Uint8Operands_Uint32Accumulators_noexpand {
385   typedef std::uint8_t OperandType;
386   typedef std::uint32_t AccumulatorType;
387   typedef KernelFormat<KernelSideFormat<CellFormat<3, 8, CellOrder::WidthMajor>, 1>,
388                        KernelSideFormat<CellFormat<3, 8, CellOrder::WidthMajor>, 1> >
389       Format;
390   static void Run(const OperandType* lhs_ptr, const OperandType* rhs_ptr, AccumulatorType* accum_ptr, int depth) {
391     asm volatile(
392       // Clear aggregators.
393       "vmov.i32 q0, #0\n"
394       "vmov.i32 q1, #0\n"
395       "vmov.i32 q2, #0\n"
396       "vmov.i32 q3, q0\n"
397       "vmov.i32 q4, q1\n"
398       "vmov.i32 q5, q2\n"
399       "vmov.i32 q6, q3\n"
400       "vmov.i32 q7, q4\n"
401       "vmov.i32 q8, q5\n"
402
403       // Loop head
404       "loop_%=:\n"
405
406       // Subtract counter.
407       "subs %[depth], %[depth], #8\n"
408
409       "vld1.8 {d18, d19, d20}, [%[rhs_ptr]]!\n"
410       "vld1.8 {d21, d22, d23}, [%[lhs_ptr]]!\n"
411       "vmull.u8 q12, d18, d21\n"
412       "vmull.u8 q13, d18, d22\n"
413       "vmull.u8 q14, d18, d23\n"
414       "vmull.u8 q15, d19, d21\n"
```

```
415         "vpadal.u16 q0, q12\n"
416         "vpadal.u16 q1, q13\n"
417         "vpadal.u16 q2, q14\n"
418         "vpadal.u16 q3, q15\n"
419         "vmull.u8 q12, d19, d22\n"
420         "vmull.u8 q13, d19, d23\n"
421         "vmull.u8 q14, d20, d21\n"
422         "vmull.u8 q15, d20, d22\n"
423         "vmull.u8 q9, d20, d23\n"
424         "vpadal.u16 q4, q12\n"
425         "vpadal.u16 q5, q13\n"
426         "vpadal.u16 q6, q14\n"
427         "vpadal.u16 q7, q15\n"
428         "vpadal.u16 q8, q9\n"
429
430         // Loop branch
431         "bne loop_%=\n"
432
433         // Horizontal reduce aggregators, step 1
434         "vpadd.u32 d0, d0, d1\n"
435         "vpadd.u32 d2, d2, d3\n"
436         "vpadd.u32 d4, d4, d5\n"
437         "vpadd.u32 d6, d6, d7\n"
438         "vpadd.u32 d8, d8, d9\n"
439         "vpadd.u32 d10, d10, d11\n"
440         "vpadd.u32 d12, d12, d13\n"
441         "vpadd.u32 d14, d14, d15\n"
442         "vpadd.u32 d16, d16, d17\n"
443
444         // Horizontal reduce aggregators, step 2
445         "vpadd.u32 d0, d0, d2\n"
446         "vpadd.u32 d1, d4, d4\n"
447         "vpadd.u32 d6, d6, d8\n"
448         "vpadd.u32 d7, d10, d10\n"
449         "vpadd.u32 d12, d12, d14\n"
450         "vpadd.u32 d13, d16, d16\n"
```

```
451
452         // Load accumulators
453         "mov r0, %[accum_ptr]\n"
454         "vld1.32 {d2}, [r0]!\n"
455         "vld1.32 {d3[0]}, [r0]!\n"
456
457         "vld1.32 {d8}, [r0]!\n"
458         "vld1.32 {d9[0]}, [r0]!\n"
459
460         "vld1.32 {d14}, [r0]!\n"
461         "vld1.32 {d15[0]}, [r0]!\n"
462
463         // Accumulate
464         "vadd.s32 q0, q0, q1\n"
465         "vadd.s32 q3, q3, q4\n"
466         "vadd.s32 q6, q6, q7\n"
467
468         // Store accumulators
469         "mov r0, %[accum_ptr]\n"
470         "vst1.32 {d0}, [r0]!\n"
471         "vst1.32 {d1[0]}, [r0]!\n"
472
473         "vst1.32 {d6}, [r0]!\n"
474         "vst1.32 {d7[0]}, [r0]!\n"
475
476         "vst1.32 {d12}, [r0]!\n"
477         "vst1.32 {d13[0]}, [r0]!\n"
478         :  // outputs
479         [lhs_ptr] "+r"(lhs_ptr), [rhs_ptr] "+r"(rhs_ptr),
480         [depth] "+r"(depth)
481         :  // inputs
482         [accum_ptr] "r"(accum_ptr)
483         :  // clobbers
484         "cc", "memory", "r0",
485         "d0", "d1", "d2", "d3", "d4", "d5", "d6", "d7", "d8", "d9", "d10",
486         "d11", "d12", "d13", "d14", "d15", "d16", "d17", "d18", "d19", "d20",
```

```
487         "d21", "d22", "d23", "d24", "d25", "d26", "d27", "d28", "d29", "d30",
488         "d31");
489     }
490 };
491
492
493 // We don't actually use int32*int32 in production. This is just an
494 // experiment to help dissociate the effect of integer-vs-float, from the
495 // effect of operands width.
496 struct NEON_32bit_GEMM_Int32_WithScalar {
497   typedef std::int32_t OperandType;
498   typedef std::int32_t AccumulatorType;
499   typedef KernelFormat<KernelSideFormat<CellFormat<4, 1, CellOrder::DepthMajor>, 3>,
500                        KernelSideFormat<CellFormat<4, 1, CellOrder::DepthMajor>, 1> >
501       Format;
502   static void Run(const OperandType* lhs_ptr, const OperandType* rhs_ptr, AccumulatorType* accum_ptr, int depth) {
503     asm volatile(
504       // Load accumulators
505       "mov r0, %[accum_ptr]\n"
506       "vld1.32 {d8, d9},   [r0]!\n"
507       "vld1.32 {d16, d17}, [r0]!\n"
508       "vld1.32 {d24, d25}, [r0]!\n"
509       "vld1.32 {d10, d11}, [r0]!\n"
510       "vld1.32 {d18, d19}, [r0]!\n"
511       "vld1.32 {d26, d27}, [r0]!\n"
512       "vld1.32 {d12, d13}, [r0]!\n"
513       "vld1.32 {d20, d21}, [r0]!\n"
514       "vld1.32 {d28, d29}, [r0]!\n"
515       "vld1.32 {d14, d15}, [r0]!\n"
516       "vld1.32 {d22, d23}, [r0]!\n"
517       "vld1.32 {d30, d31}, [r0]!\n"
518
519       "loop_%=:\n"
520
521       // Load 1 Rhs cell of size 1x4
522       "vld1.32 {d0, d1}, [%[rhs_ptr]]!\n"
```

```
523
524         // Load 3 Lhs cells of size 4x1 each
525         "vld1.32 {d2, d3}, [%[lhs_ptr]]!\n"
526         "vld1.32 {d4, d5}, [%[lhs_ptr]]!\n"
527         "vld1.32 {d6, d7}, [%[lhs_ptr]]!\n"
528
529         // Multiply-accumulate
530         "vmla.s32 q4, q1, d0[0]\n"
531         "vmla.s32 q5, q1, d0[1]\n"
532         "vmla.s32 q6, q1, d1[0]\n"
533         "vmla.s32 q7, q1, d1[1]\n"
534         "vmla.s32 q8, q2, d0[0]\n"
535         "vmla.s32 q9, q2, d0[1]\n"
536         "vmla.s32 q10, q2, d1[0]\n"
537         "vmla.s32 q11, q2, d1[1]\n"
538         "vmla.s32 q12, q3, d0[0]\n"
539         "vmla.s32 q13, q3, d0[1]\n"
540         "vmla.s32 q14, q3, d1[0]\n"
541         "vmla.s32 q15, q3, d1[1]\n"
542
543         // Loop. Decrement loop index (depth) by 1, since we just handled 1
544         // level of depth.
545         "subs %[depth], #1\n"
546         "bne loop_%=\n"
547
548         // Store accumulators
549         "mov r0, %[accum_ptr]\n"
550         "vst1.32 {d8, d9},   [r0]!\n"
551         "vst1.32 {d16, d17}, [r0]!\n"
552         "vst1.32 {d24, d25}, [r0]!\n"
553         "vst1.32 {d10, d11}, [r0]!\n"
554         "vst1.32 {d18, d19}, [r0]!\n"
555         "vst1.32 {d26, d27}, [r0]!\n"
556         "vst1.32 {d12, d13}, [r0]!\n"
557         "vst1.32 {d20, d21}, [r0]!\n"
558         "vst1.32 {d28, d29}, [r0]!\n"
```

```
559         "vst1.32 {d14, d15}, [r0]!\n"
560         "vst1.32 {d22, d23}, [r0]!\n"
561         "vst1.32 {d30, d31}, [r0]!\n"
562         :  // outputs
563         [lhs_ptr] "+r"(lhs_ptr), [rhs_ptr] "+r"(rhs_ptr),
564         [depth] "+r"(depth)
565         :  // inputs
566         [accum_ptr] "r"(accum_ptr)
567         :  // clobbers
568         "cc", "memory", "r0",
569         "d0", "d1", "d2", "d3", "d4", "d5", "d6", "d7", "d8", "d9", "d10",
570         "d11", "d12", "d13", "d14", "d15", "d16", "d17", "d18", "d19", "d20",
571         "d21", "d22", "d23", "d24", "d25", "d26", "d27", "d28", "d29", "d30",
572         "d31");
573     }
574 };
575
576 // Not very efficient kernel, just an experiment to see what we can do
577 // without using NEON multiply-with-scalar instructions.
578 struct NEON_32bit_GEMM_Float32_MLA_WithVectorDuplicatingScalar {
579   typedef float OperandType;
580   typedef float AccumulatorType;
581   typedef KernelFormat<KernelSideFormat<CellFormat<4, 1, CellOrder::DepthMajor>, 3>,
582                        KernelSideFormat<CellFormat<4, 1, CellOrder::DepthMajor>, 1> >
583       Format;
584   static void Run(const OperandType* lhs_ptr, const OperandType* rhs_ptr, AccumulatorType* accum_ptr, int depth) {
585     asm volatile(
586       // Load accumulators
587       "mov r0, %[accum_ptr]\n"
588       "vld1.32 {d8, d9},   [r0]!\n"
589       "vld1.32 {d16, d17}, [r0]!\n"
590       "vld1.32 {d24, d25}, [r0]!\n"
591       "vld1.32 {d10, d11}, [r0]!\n"
592       "vld1.32 {d18, d19}, [r0]!\n"
593       "vld1.32 {d26, d27}, [r0]!\n"
594       "vld1.32 {d12, d13}, [r0]!\n"
```

```
595         "vld1.32 {d20, d21}, [r0]!\n"
596         "vld1.32 {d28, d29}, [r0]!\n"
597         "vld1.32 {d14, d15}, [r0]!\n"
598         "vld1.32 {d22, d23}, [r0]!\n"
599         "vld1.32 {d30, d31}, [r0]!\n"
600
601         "loop_%=:\n"
602
603         // Load 3 Lhs cells of size 4x1 each
604         "vld1.32 {d2, d3}, [%[lhs_ptr]]!\n"
605         "vld1.32 {d4, d5}, [%[lhs_ptr]]!\n"
606         "vld1.32 {d6, d7}, [%[lhs_ptr]]!\n"
607
608         // Multiply-accumulate
609         "vld1.32 {d0[], d1[]}, [%[rhs_ptr]]!\n"
610         "vmla.f32 q4, q1, q0\n"
611         "vmla.f32 q8, q2, q0\n"
612         "vmla.f32 q12, q3, q0\n"
613         "vld1.32 {d0[], d1[]}, [%[rhs_ptr]]!\n"
614         "vmla.f32 q5, q1, q0\n"
615         "vmla.f32 q9, q2, q0\n"
616         "vmla.f32 q13, q3, q0\n"
617         "vld1.32 {d0[], d1[]}, [%[rhs_ptr]]!\n"
618         "vmla.f32 q6, q1, q0\n"
619         "vmla.f32 q10, q2, q0\n"
620         "vmla.f32 q14, q3, q0\n"
621         "vld1.32 {d0[], d1[]}, [%[rhs_ptr]]!\n"
622         "vmla.f32 q7, q1, q0\n"
623         "vmla.f32 q11, q2, q0\n"
624         "vmla.f32 q15, q3, q0\n"
625
626         // Loop. Decrement loop index (depth) by 1, since we just handled 1
627         // level of depth.
628         "subs %[depth], #1\n"
629         "bne loop_%=\n"
630
```

```
631         // Store accumulators
632         "mov r0, %[accum_ptr]\n"
633         "vst1.32 {d8, d9},   [r0]!\n"
634         "vst1.32 {d16, d17}, [r0]!\n"
635         "vst1.32 {d24, d25}, [r0]!\n"
636         "vst1.32 {d10, d11}, [r0]!\n"
637         "vst1.32 {d18, d19}, [r0]!\n"
638         "vst1.32 {d26, d27}, [r0]!\n"
639         "vst1.32 {d12, d13}, [r0]!\n"
640         "vst1.32 {d20, d21}, [r0]!\n"
641         "vst1.32 {d28, d29}, [r0]!\n"
642         "vst1.32 {d14, d15}, [r0]!\n"
643         "vst1.32 {d22, d23}, [r0]!\n"
644         "vst1.32 {d30, d31}, [r0]!\n"
645         :  // outputs
646         [lhs_ptr] "+r"(lhs_ptr), [rhs_ptr] "+r"(rhs_ptr),
647         [depth] "+r"(depth)
648         :  // inputs
649         [accum_ptr] "r"(accum_ptr)
650         :  // clobbers
651         "cc", "memory", "r0",
652         "d0", "d1", "d2", "d3", "d4", "d5", "d6", "d7", "d8", "d9", "d10",
653         "d11", "d12", "d13", "d14", "d15", "d16", "d17", "d18", "d19", "d20",
654         "d21", "d22", "d23", "d24", "d25", "d26", "d27", "d28", "d29", "d30",
655         "d31");
656     }
657 };
658
659 // Not very efficient kernel, just an experiment to see what we can do
660 // without using NEON multiply-with-scalar instructions.
661 // This variant is relevant as on ARMv7 FMA does not have a with-scalar variant.
662 struct NEON_32bit_GEMM_Float32_FMA_WithVectorDuplicatingScalar {
663   typedef float OperandType;
664   typedef float AccumulatorType;
665   typedef KernelFormat<KernelSideFormat<CellFormat<4, 1, CellOrder::DepthMajor>, 3>,
666                        KernelSideFormat<CellFormat<4, 1, CellOrder::DepthMajor>, 1> >
```

```
667         Format;
668     static void Run(const OperandType* lhs_ptr, const OperandType* rhs_ptr, AccumulatorType* accum_ptr, int depth) {
669       asm volatile(
670         // Load accumulators
671         "mov r0, %[accum_ptr]\n"
672         "vld1.32 {d8, d9},   [r0]!\n"
673         "vld1.32 {d16, d17}, [r0]!\n"
674         "vld1.32 {d24, d25}, [r0]!\n"
675         "vld1.32 {d10, d11}, [r0]!\n"
676         "vld1.32 {d18, d19}, [r0]!\n"
677         "vld1.32 {d26, d27}, [r0]!\n"
678         "vld1.32 {d12, d13}, [r0]!\n"
679         "vld1.32 {d20, d21}, [r0]!\n"
680         "vld1.32 {d28, d29}, [r0]!\n"
681         "vld1.32 {d14, d15}, [r0]!\n"
682         "vld1.32 {d22, d23}, [r0]!\n"
683         "vld1.32 {d30, d31}, [r0]!\n"
684
685         "loop_%=:\n"
686
687         // Load 3 Lhs cells of size 4x1 each
688         "vld1.32 {d2, d3}, [%[lhs_ptr]]!\n"
689         "vld1.32 {d4, d5}, [%[lhs_ptr]]!\n"
690         "vld1.32 {d6, d7}, [%[lhs_ptr]]!\n"
691
692         // Multiply-accumulate
693         "vld1.32 {d0[], d1[]}, [%[rhs_ptr]]!\n"
694         "vfma.f32 q4, q1, q0\n"
695         "vfma.f32 q8, q2, q0\n"
696         "vfma.f32 q12, q3, q0\n"
697         "vld1.32 {d0[], d1[]}, [%[rhs_ptr]]!\n"
698         "vfma.f32 q5, q1, q0\n"
699         "vfma.f32 q9, q2, q0\n"
700         "vfma.f32 q13, q3, q0\n"
701         "vld1.32 {d0[], d1[]}, [%[rhs_ptr]]!\n"
702         "vfma.f32 q6, q1, q0\n"
```

```
703         "vfma.f32 q10, q2, q0\n"
704         "vfma.f32 q14, q3, q0\n"
705         "vld1.32 {d0[], d1[]}, [%[rhs_ptr]]!\n"
706         "vfma.f32 q7, q1, q0\n"
707         "vfma.f32 q11, q2, q0\n"
708         "vfma.f32 q15, q3, q0\n"
709
710         // Loop. Decrement loop index (depth) by 1, since we just handled 1
711         // level of depth.
712         "subs %[depth], #1\n"
713         "bne loop_%=\n"
714
715         // Store accumulators
716         "mov r0, %[accum_ptr]\n"
717         "vst1.32 {d8, d9},   [r0]!\n"
718         "vst1.32 {d16, d17}, [r0]!\n"
719         "vst1.32 {d24, d25}, [r0]!\n"
720         "vst1.32 {d10, d11}, [r0]!\n"
721         "vst1.32 {d18, d19}, [r0]!\n"
722         "vst1.32 {d26, d27}, [r0]!\n"
723         "vst1.32 {d12, d13}, [r0]!\n"
724         "vst1.32 {d20, d21}, [r0]!\n"
725         "vst1.32 {d28, d29}, [r0]!\n"
726         "vst1.32 {d14, d15}, [r0]!\n"
727         "vst1.32 {d22, d23}, [r0]!\n"
728         "vst1.32 {d30, d31}, [r0]!\n"
729         :  // outputs
730         [lhs_ptr] "+r"(lhs_ptr), [rhs_ptr] "+r"(rhs_ptr),
731         [depth] "+r"(depth)
732         :  // inputs
733         [accum_ptr] "r"(accum_ptr)
734         :  // clobbers
735         "cc", "memory", "r0",
736         "d0", "d1", "d2", "d3", "d4", "d5", "d6", "d7", "d8", "d9", "d10",
737         "d11", "d12", "d13", "d14", "d15", "d16", "d17", "d18", "d19", "d20",
738         "d21", "d22", "d23", "d24", "d25", "d26", "d27", "d28", "d29", "d30",
```

```
739         "d31");
740     }
741   };
742
743   // This is the "most natural" kernel, using NEON multiply-with-scalar instructions.
744   struct NEON_32bit_GEMM_Float32_MLA_WithScalar {
745     typedef float OperandType;
746     typedef float AccumulatorType;
747     typedef KernelFormat<KernelSideFormat<CellFormat<4, 1, CellOrder::DepthMajor>, 3>,
748                          KernelSideFormat<CellFormat<4, 1, CellOrder::DepthMajor>, 1> >
749         Format;
750     static void Run(const OperandType* lhs_ptr, const OperandType* rhs_ptr, AccumulatorType* accum_ptr, int depth) {
751       asm volatile(
752         // Load accumulators
753         "mov r0, %[accum_ptr]\n"
754         "vld1.32 {d8, d9},   [r0]!\n"
755         "vld1.32 {d16, d17}, [r0]!\n"
756         "vld1.32 {d24, d25}, [r0]!\n"
757         "vld1.32 {d10, d11}, [r0]!\n"
758         "vld1.32 {d18, d19}, [r0]!\n"
759         "vld1.32 {d26, d27}, [r0]!\n"
760         "vld1.32 {d12, d13}, [r0]!\n"
761         "vld1.32 {d20, d21}, [r0]!\n"
762         "vld1.32 {d28, d29}, [r0]!\n"
763         "vld1.32 {d14, d15}, [r0]!\n"
764         "vld1.32 {d22, d23}, [r0]!\n"
765         "vld1.32 {d30, d31}, [r0]!\n"
766
767         "loop_%=:\n"
768
769         // Load 1 Rhs cell of size 1x4
770         "vld1.32 {d0, d1}, [%[rhs_ptr]]!\n"
771
772         // Load 3 Lhs cells of size 4x1 each
773         "vld1.32 {d2, d3}, [%[lhs_ptr]]!\n"
774         "vld1.32 {d4, d5}, [%[lhs_ptr]]!\n"
```

```
775         "vld1.32 {d6, d7}, [%[lhs_ptr]]!\n"
776
777         // Multiply-accumulate
778         "vmla.f32 q4, q1, d0[0]\n"
779         "vmla.f32 q5, q1, d0[1]\n"
780         "vmla.f32 q6, q1, d1[0]\n"
781         "vmla.f32 q7, q1, d1[1]\n"
782         "vmla.f32 q8, q2, d0[0]\n"
783         "vmla.f32 q9, q2, d0[1]\n"
784         "vmla.f32 q10, q2, d1[0]\n"
785         "vmla.f32 q11, q2, d1[1]\n"
786         "vmla.f32 q12, q3, d0[0]\n"
787         "vmla.f32 q13, q3, d0[1]\n"
788         "vmla.f32 q14, q3, d1[0]\n"
789         "vmla.f32 q15, q3, d1[1]\n"
790
791         // Loop. Decrement loop index (depth) by 1, since we just handled 1
792         // level of depth.
793         "subs %[depth], #1\n"
794         "bne loop_%=\n"
795
796         // Store accumulators
797         "mov r0, %[accum_ptr]\n"
798         "vst1.32 {d8, d9},   [r0]!\n"
799         "vst1.32 {d16, d17}, [r0]!\n"
800         "vst1.32 {d24, d25}, [r0]!\n"
801         "vst1.32 {d10, d11}, [r0]!\n"
802         "vst1.32 {d18, d19}, [r0]!\n"
803         "vst1.32 {d26, d27}, [r0]!\n"
804         "vst1.32 {d12, d13}, [r0]!\n"
805         "vst1.32 {d20, d21}, [r0]!\n"
806         "vst1.32 {d28, d29}, [r0]!\n"
807         "vst1.32 {d14, d15}, [r0]!\n"
808         "vst1.32 {d22, d23}, [r0]!\n"
809         "vst1.32 {d30, d31}, [r0]!\n"
810         :  // outputs
```

```
811         [lhs_ptr] "+r"(lhs_ptr), [rhs_ptr] "+r"(rhs_ptr),
812         [depth] "+r"(depth)
813         :  // inputs
814         [accum_ptr] "r"(accum_ptr)
815         :  // clobbers
816         "cc", "memory", "r0",
817         "d0", "d1", "d2", "d3", "d4", "d5", "d6", "d7", "d8", "d9", "d10",
818         "d11", "d12", "d13", "d14", "d15", "d16", "d17", "d18", "d19", "d20",
819         "d21", "d22", "d23", "d24", "d25", "d26", "d27", "d28", "d29", "d30",
820         "d31");
821     }
822 };
823
824 // Faster kernel contributed by ARM in 64bit form
825 // (see NEON_64bit_GEMM_Float32_WithScalar_A53) then ported to 32bit code.
826 // Tuned for A53.
827 struct NEON_32bit_GEMM_Float32_WithScalar_A53 {
828   typedef float OperandType;
829   typedef float AccumulatorType;
830   typedef KernelFormat<KernelSideFormat<CellFormat<4, 1, CellOrder::DepthMajor>, 3>,
831                        KernelSideFormat<CellFormat<4, 1, CellOrder::DepthMajor>, 1> >
832       Format;
833   static void Run(const OperandType* lhs_ptr, const OperandType* rhs_ptr, AccumulatorType* accum_ptr, int depth) {
834     asm volatile(
835       // Load accumulators
836       "mov r0, %[accum_ptr]\n"
837       "vld1.32 {d8, d9},   [r0]!\n"
838       "vld1.32 {d16, d17}, [r0]!\n"
839       "vld1.32 {d24, d25}, [r0]!\n"
840       "vld1.32 {d10, d11}, [r0]!\n"
841       "vld1.32 {d18, d19}, [r0]!\n"
842       "vld1.32 {d26, d27}, [r0]!\n"
843       "vld1.32 {d12, d13}, [r0]!\n"
844       "vld1.32 {d20, d21}, [r0]!\n"
845       "vld1.32 {d28, d29}, [r0]!\n"
846       "vld1.32 {d14, d15}, [r0]!\n"
```

```
847         "vld1.32 {d22, d23}, [r0]!\n"
848         "vld1.32 {d30, d31}, [r0]!\n"
849
850         // Overview of register layout:
851         //
852         // A 1x4 cell of Rhs is stored in d0--d1 (q0).
853         // A 12x1 block of 3 4x1 cells Lhs is stored in d2--d7
854         // (q1--q3).
855         // A 12x4 block of accumulators is stored in q4--q15.
856         //
857         //                  +-----+-----+-----+-----+
858         //             Rhs  |d0[0]|d0[1]|d1[0]|d1[1]|
859         //                  +-----+-----+-----+-----+
860         //
861         //                  |     |     |     |     |
862         //
863         //  Lhs             |     |     |     |     |
864         //
865         //  +--+- - - - - -  +-----+-----+-----+-----+
866         //  |d2|             | q4  | q5  | q6  | q7  |
867         //  |d2|             | q4  | q5  | q6  | q7  |
868         //  |d3|             | q4  | q5  | q6  | q7  |
869         //  |d3|             | q4  | q5  | q6  | q7  |
870         //  +--+- - - - - -  +-----+-----+-----+-----+
871         //  |d4|             | q8  | q9  | q10 | q11 |
872         //  |d4|             | q8  | q9  | q10 | q11 |
873         //  |d5|             | q8  | q9  | q10 | q11 |
874         //  |d5|             | q8  | q9  | q10 | q11 |
875         //  +--+ - - - - - - +-----+-----+-----+-----+
876         //  |d6|             | q12 | q13 | q14 | q15 |
877         //  |d6|             | q12 | q13 | q14 | q15 |
878         //  |d7|             | q12 | q13 | q14 | q15 |
879         //  |d7|             | q12 | q13 | q14 | q15 |
880         //  +--+- - - - - -  +-----+-----+-----+-----+
881         //
882         //                           Accumulator
```

```
883
884         // Load Rhs cell
885         "vldr d0, [%[rhs_ptr]]\n"
886         "ldr r2, [%[rhs_ptr], #8]\n"
887         "ldr r3, [%[rhs_ptr], #12]\n"
888
889         // Load 1st Lhs Cell
890         "vld1.32 {d2, d3}, [%[lhs_ptr]]\n"
891
892         "loop_%=:\n" // Loop head
893
894         "vldr d4, [%[lhs_ptr], #16]\n" // Load 1st half of 2nd Lhs cell
895         "vmov d1, r2, r3\n"            // Prepare 2nd half of Rhs cell
896         "vmla.f32 q4, q1, d0[0]\n"     // Multiply 1st Lhs cell with column 0
897         "ldr r2, [%[lhs_ptr], #24]\n"  // Load 2nd half of 2nd Lhs cell, part 1
898         "vmla.f32 q5, q1, d0[1]\n"     // Multiply 1st Lhs cell with column 1
899         "ldr r3, [%[lhs_ptr], #28]\n"  // Load 2nd half of 2nd Lhs cell, part 2
900         "vmla.f32 q6, q1, d1[0]\n"     // Multiply 1st Lhs cell with column 2
901         "subs %[depth], #1\n"
902
903         "vldr d6, [%[lhs_ptr], #32]\n" // Load 1st half of 3rd Lhs cell
904         "vmov d5, r2, r3\n"            // Prepare 2nd half of 2nd Lhs cell
905         "vmla.f32 q7, q1, d1[1]\n"     // Multiply 1st Lhs cell with column 3
906         "ldr r2, [%[lhs_ptr], #40]\n"  // Load 2nd half of 3rd Lhs cell, part 1
907         "vmla.f32 q8, q2, d0[0]\n"     // Multiply 2nd Lhs cell with column 0
908         "ldr r3, [%[lhs_ptr], #44]\n"  // Load 2nd half of 3rd Lhs cell, part 2
909         "vmla.f32 q9, q2, d0[1]\n"     // Multiply 2nd Lhs cell with column 1
910         "add %[rhs_ptr], %[rhs_ptr], #16\n" // Move forward by 1 Rhs cell
911
912         "vldr d2, [%[lhs_ptr], #48]\n" // Load 1st half of 1st Lhs cell of next iteration
913         "vmov d7, r2, r3\n"            // Prepare 2nd half of 3rd Lhs cell
914         "vmla.f32 q10, q2, d1[0]\n"    // Multiply 2nd Lhs cell with column 2
915         "ldr r2, [%[lhs_ptr], #56]\n"  // Load 2nd half of 1st Lhs cell of next iter, part 1
916         "vmla.f32 q12, q3, d0[0]\n"    // Multiply 3rd Lhs cell with column 0
917         "ldr r3, [%[lhs_ptr], #60]\n"  // Load 2nd half of 1st Lhs cell of next iter, part 2
918         "vmla.f32 q13, q3, d0[1]\n"    // Multiply 3rd Lhs cell with column 1
```

```
919         "add %[lhs_ptr], %[lhs_ptr], #48\n" // Move forward by 3 Lhs cells
920
921         "vldr d0, [%[rhs_ptr]]\n" // Load 1st half of Rhs cell of next iteration
922         "vmov d3, r2, r3\n"            // Prepare 2nd half of 1st Lhs cell of next iteration
923         "vmla.f32 q11, q2, d1[1]\n"    // Multiply 2nd Lhs cell with column 3
924         "ldr r2, [%[rhs_ptr], #8]\n"  // Load 2nd half of Rhs cell of next iteration, part 1
925         "vmla.f32 q14, q3, d1[0]\n"    // Multiply 3rd Lhs cell with column 2
926         "ldr r3, [%[rhs_ptr], #12]\n"  // Load 2nd half of Rhs cell of next iteration, part 2
927         "vmla.f32 q15, q3, d1[1]\n"    // Multiply 3rd Lhs cell with column 3
928
929         // Loop branch.  This will dual issue in fmla cycle 3 of the 4th block.
930         "bne loop_%=\n"
931
932         // Store accumulators
933         "mov r0, %[accum_ptr]\n"
934         "vst1.32 {d8, d9},   [r0]!\n"
935         "vst1.32 {d16, d17}, [r0]!\n"
936         "vst1.32 {d24, d25}, [r0]!\n"
937         "vst1.32 {d10, d11}, [r0]!\n"
938         "vst1.32 {d18, d19}, [r0]!\n"
939         "vst1.32 {d26, d27}, [r0]!\n"
940         "vst1.32 {d12, d13}, [r0]!\n"
941         "vst1.32 {d20, d21}, [r0]!\n"
942         "vst1.32 {d28, d29}, [r0]!\n"
943         "vst1.32 {d14, d15}, [r0]!\n"
944         "vst1.32 {d22, d23}, [r0]!\n"
945         "vst1.32 {d30, d31}, [r0]!\n"
946         :  // outputs
947         [lhs_ptr] "+r"(lhs_ptr), [rhs_ptr] "+r"(rhs_ptr),
948         [depth] "+r"(depth)
949         :  // inputs
950         [accum_ptr] "r"(accum_ptr)
951         :  // clobbers
952         "cc", "memory", "r0", "r2", "r3",
953         "d0", "d1", "d2", "d3", "d4", "d5", "d6", "d7", "d8", "d9", "d10",
954         "d11", "d12", "d13", "d14", "d15", "d16", "d17", "d18", "d19", "d20",
```

```
955         "d21", "d22", "d23", "d24", "d25", "d26", "d27", "d28", "d29", "d30",
956         "d31");
957     }
958 };
959
960 struct NEON_32bit_GEMM_Float32_WithScalar_A53_depth2 {
961     typedef float OperandType;
962     typedef float AccumulatorType;
963     typedef KernelFormat<KernelSideFormat<CellFormat<4, 2, CellOrder::DepthMajor>, 3>,
964                          KernelSideFormat<CellFormat<4, 2, CellOrder::DepthMajor>, 1> >
965         Format;
966       static void Run(const OperandType* lhs_ptr, const OperandType* rhs_ptr, AccumulatorType* accum_ptr, int depth) {
967       asm volatile(
968         // Load accumulators
969         "mov r0, %[accum_ptr]\n"
970         "vld1.32 {d8, d9},   [r0]!\n"
971         "vld1.32 {d16, d17}, [r0]!\n"
972         "vld1.32 {d24, d25}, [r0]!\n"
973         "vld1.32 {d10, d11}, [r0]!\n"
974         "vld1.32 {d18, d19}, [r0]!\n"
975         "vld1.32 {d26, d27}, [r0]!\n"
976         "vld1.32 {d12, d13}, [r0]!\n"
977         "vld1.32 {d20, d21}, [r0]!\n"
978         "vld1.32 {d28, d29}, [r0]!\n"
979         "vld1.32 {d14, d15}, [r0]!\n"
980         "vld1.32 {d22, d23}, [r0]!\n"
981         "vld1.32 {d30, d31}, [r0]!\n"
982
983         // Overview of register layout:
984         //
985         // A 1x4 cell of Rhs is stored in d0--d1 (q0).
986         // A 12x1 block of 3 4x1 cells Lhs is stored in d2--d7
987         // (q1--q3).
988         // A 12x4 block of accumulators is stored in q4--q15.
989         //
990         //                  +-----+-----+-----+-----+
```

```
991         //             Rhs   |d0[0]|d0[1]|d1[0]|d1[1]|
992         //                   +-----+-----+-----+-----+
993         //
994         //                   |     |     |     |     |
995         //
996         //  Lhs              |     |     |     |     |
997         //
998         //  +--+- - - - - -  +-----+-----+-----+-----+
999         //  |d2|             | q4  | q5  | q6  | q7  |
1000        //  |d2|             | q4  | q5  | q6  | q7  |
1001        //  |d3|             | q4  | q5  | q6  | q7  |
1002        //  |d3|             | q4  | q5  | q6  | q7  |
1003        //  +--+- - - - - -  +-----+-----+-----+-----+
1004        //  |d4|             | q8  | q9  | q10 | q11 |
1005        //  |d4|             | q8  | q9  | q10 | q11 |
1006        //  |d5|             | q8  | q9  | q10 | q11 |
1007        //  |d5|             | q8  | q9  | q10 | q11 |
1008        //  +--+ - - - - - - +-----+-----+-----+-----+
1009        //  |d6|             | q12 | q13 | q14 | q15 |
1010        //  |d6|             | q12 | q13 | q14 | q15 |
1011        //  |d7|             | q12 | q13 | q14 | q15 |
1012        //  |d7|             | q12 | q13 | q14 | q15 |
1013        //  +--+- - - - - -  +-----+-----+-----+-----+
1014        //
1015        //                            Accumulator
1016
1017        // Load Rhs cell
1018        "vldr d0, [%[rhs_ptr]]\n"
1019        "ldr r2, [%[rhs_ptr], #8]\n"
1020        "ldr r3, [%[rhs_ptr], #12]\n"
1021
1022        // Load 1st Lhs Cell
1023        "vld1.32 {d2, d3}, [%[lhs_ptr]]\n"
1024
1025        "loop_%=:\n" // Loop head - handling 2 levels of depth at once
1026
```

```
1027        // Level of depth 1
1028
1029        "vldr d4, [%[lhs_ptr], #32]\n" // Load 1st half of 2nd Lhs cell
1030        "vmov d1, r2, r3\n"            // Prepare 2nd half of Rhs cell
1031        "vmla.f32 q4, q1, d0[0]\n"     // Multiply 1st Lhs cell with column 0
1032        "ldr r2, [%[lhs_ptr], #40]\n"  // Load 2nd half of 2nd Lhs cell, part 1
1033        "vmla.f32 q5, q1, d0[1]\n"     // Multiply 1st Lhs cell with column 1
1034        "ldr r3, [%[lhs_ptr], #44]\n"  // Load 2nd half of 2nd Lhs cell, part 2
1035        "vmla.f32 q6, q1, d1[0]\n"     // Multiply 1st Lhs cell with column 2
1036
1037        "vldr d6, [%[lhs_ptr], #64]\n" // Load 1st half of 3rd Lhs cell
1038        "vmov d5, r2, r3\n"            // Prepare 2nd half of 2nd Lhs cell
1039        "vmla.f32 q7, q1, d1[1]\n"     // Multiply 1st Lhs cell with column 3
1040        "ldr r2, [%[lhs_ptr], #72]\n"  // Load 2nd half of 3rd Lhs cell, part 1
1041        "vmla.f32 q8, q2, d0[0]\n"     // Multiply 2nd Lhs cell with column 0
1042        "ldr r3, [%[lhs_ptr], #76]\n"  // Load 2nd half of 3rd Lhs cell, part 2
1043        "vmla.f32 q9, q2, d0[1]\n"     // Multiply 2nd Lhs cell with column 1
1044
1045        "vldr d2, [%[lhs_ptr], #16]\n" // Load 1st half of 1st Lhs cell of next iteration
1046        "vmov d7, r2, r3\n"            // Prepare 2nd half of 3rd Lhs cell
1047        "vmla.f32 q10, q2, d1[0]\n"    // Multiply 2nd Lhs cell with column 2
1048        "ldr r2, [%[lhs_ptr], #24]\n"  // Load 2nd half of 1st Lhs cell of next iter, part 1
1049        "vmla.f32 q12, q3, d0[0]\n"    // Multiply 3rd Lhs cell with column 0
1050        "ldr r3, [%[lhs_ptr], #28]\n"  // Load 2nd half of 1st Lhs cell of next iter, part 2
1051        "vmla.f32 q13, q3, d0[1]\n"    // Multiply 3rd Lhs cell with column 1
1052
1053        "vldr d0, [%[rhs_ptr], #16]\n" // Load 1st half of Rhs cell of next iteration
1054        "vmov d3, r2, r3\n"            // Prepare 2nd half of 1st Lhs cell of next iteration
1055        "vmla.f32 q11, q2, d1[1]\n"    // Multiply 2nd Lhs cell with column 3
1056        "ldr r2, [%[rhs_ptr], #24]\n"  // Load 2nd half of Rhs cell of next iteration, part 1
1057        "vmla.f32 q14, q3, d1[0]\n"    // Multiply 3rd Lhs cell with column 2
1058        "ldr r3, [%[rhs_ptr], #28]\n"  // Load 2nd half of Rhs cell of next iteration, part 2
1059        "vmla.f32 q15, q3, d1[1]\n"    // Multiply 3rd Lhs cell with column 3
1060
1061        // Level of depth 2
1062
```

```
1063        "loop_second_unrolled_iter_%=:\n"
1064
1065        "vldr d4, [%[lhs_ptr], #48]\n" // Load 1st half of 2nd Lhs cell
1066        "vmov d1, r2, r3\n"            // Prepare 2nd half of Rhs cell
1067        "vmla.f32 q4, q1, d0[0]\n"     // Multiply 1st Lhs cell with column 0
1068        "ldr r2, [%[lhs_ptr], #56]\n"  // Load 2nd half of 2nd Lhs cell, part 1
1069        "vmla.f32 q5, q1, d0[1]\n"     // Multiply 1st Lhs cell with column 1
1070        "ldr r3, [%[lhs_ptr], #60]\n"  // Load 2nd half of 2nd Lhs cell, part 2
1071        "vmla.f32 q6, q1, d1[0]\n"     // Multiply 1st Lhs cell with column 2
1072        "subs %[depth], #2\n"          // Decrement depth counter
1073
1074        "vldr d6, [%[lhs_ptr], #80]\n" // Load 1st half of 3rd Lhs cell
1075        "vmov d5, r2, r3\n"            // Prepare 2nd half of 2nd Lhs cell
1076        "vmla.f32 q7, q1, d1[1]\n"     // Multiply 1st Lhs cell with column 3
1077        "ldr r2, [%[lhs_ptr], #88]\n"  // Load 2nd half of 3rd Lhs cell, part 1
1078        "vmla.f32 q8, q2, d0[0]\n"     // Multiply 2nd Lhs cell with column 0
1079        "ldr r3, [%[lhs_ptr], #92]\n"  // Load 2nd half of 3rd Lhs cell, part 2
1080        "vmla.f32 q9, q2, d0[1]\n"     // Multiply 2nd Lhs cell with column 1
1081        "add %[rhs_ptr], %[rhs_ptr], #32\n" // Move forward by 1 Rhs cell
1082
1083        "vldr d2, [%[lhs_ptr], #96]\n" // Load 1st half of 1st Lhs cell of next iteration
1084        "vmov d7, r2, r3\n"            // Prepare 2nd half of 3rd Lhs cell
1085        "vmla.f32 q10, q2, d1[0]\n"    // Multiply 2nd Lhs cell with column 2
1086        "ldr r2, [%[lhs_ptr], #104]\n"  // Load 2nd half of 1st Lhs cell of next iter, part 1
1087        "vmla.f32 q12, q3, d0[0]\n"    // Multiply 3rd Lhs cell with column 0
1088        "ldr r3, [%[lhs_ptr], #108]\n"  // Load 2nd half of 1st Lhs cell of next iter, part 2
1089        "vmla.f32 q13, q3, d0[1]\n"    // Multiply 3rd Lhs cell with column 1
1090        "add %[lhs_ptr], %[lhs_ptr], #96\n" // Move forward by 3 Lhs cells
1091
1092        "vldr d0, [%[rhs_ptr]]\n" // Load 1st half of Rhs cell of next iteration
1093        "vmov d3, r2, r3\n"            // Prepare 2nd half of 1st Lhs cell of next iteration
1094        "vmla.f32 q11, q2, d1[1]\n"    // Multiply 2nd Lhs cell with column 3
1095        "ldr r2, [%[rhs_ptr], #8]\n"  // Load 2nd half of Rhs cell of next iteration, part 1
1096        "vmla.f32 q14, q3, d1[0]\n"    // Multiply 3rd Lhs cell with column 2
1097        "ldr r3, [%[rhs_ptr], #12]\n"  // Load 2nd half of Rhs cell of next iteration, part 2
1098        "vmla.f32 q15, q3, d1[1]\n"    // Multiply 3rd Lhs cell with column 3
```

```
1099
1100        // Loop branch.  This will dual issue in fmla cycle 3 of the 4th block.
1101        "bne loop_%=\n"
1102
1103        // Store accumulators
1104        "mov r0, %[accum_ptr]\n"
1105        "vst1.32 {d8, d9},   [r0]!\n"
1106        "vst1.32 {d16, d17}, [r0]!\n"
1107        "vst1.32 {d24, d25}, [r0]!\n"
1108        "vst1.32 {d10, d11}, [r0]!\n"
1109        "vst1.32 {d18, d19}, [r0]!\n"
1110        "vst1.32 {d26, d27}, [r0]!\n"
1111        "vst1.32 {d12, d13}, [r0]!\n"
1112        "vst1.32 {d20, d21}, [r0]!\n"
1113        "vst1.32 {d28, d29}, [r0]!\n"
1114        "vst1.32 {d14, d15}, [r0]!\n"
1115        "vst1.32 {d22, d23}, [r0]!\n"
1116        "vst1.32 {d30, d31}, [r0]!\n"
1117        :  // outputs
1118        [lhs_ptr] "+r"(lhs_ptr), [rhs_ptr] "+r"(rhs_ptr),
1119        [depth] "+r"(depth)
1120        :  // inputs
1121        [accum_ptr] "r"(accum_ptr)
1122        :  // clobbers
1123        "cc", "memory", "r0", "r2", "r3",
1124        "d0", "d1", "d2", "d3", "d4", "d5", "d6", "d7", "d8", "d9", "d10",
1125        "d11", "d12", "d13", "d14", "d15", "d16", "d17", "d18", "d19", "d20",
1126        "d21", "d22", "d23", "d24", "d25", "d26", "d27", "d28", "d29", "d30",
1127        "d31");
1128    }
1129  };
1130
1131  // This rotating variant performs well when permutations (vext) can be dual-issued
1132  // with arithmetic instructions.
1133  struct NEON_32bit_GEMM_Float32_MLA_Rotating {
1134    typedef float OperandType;
```

```
1135    typedef float AccumulatorType;
1136    typedef KernelFormat<KernelSideFormat<CellFormat<4, 1, CellOrder::DepthMajor>, 3>,
1137                         KernelSideFormat<CellFormat<4, 1, CellOrder::DepthMajor>, 1> >
1138        Format;
1139    static void Run(const OperandType* lhs_ptr, const OperandType* rhs_ptr, AccumulatorType* accum_ptr, int depth) {
1140      asm volatile(
1141        // Load accumulators
1142        "mov r0, %[accum_ptr]\n"
1143        "vld1.32 {d8, d9},   [r0]!\n"
1144        "vld1.32 {d16, d17}, [r0]!\n"
1145        "vld1.32 {d24, d25}, [r0]!\n"
1146        "vld1.32 {d10, d11}, [r0]!\n"
1147        "vld1.32 {d18, d19}, [r0]!\n"
1148        "vld1.32 {d26, d27}, [r0]!\n"
1149        "vld1.32 {d12, d13}, [r0]!\n"
1150        "vld1.32 {d20, d21}, [r0]!\n"
1151        "vld1.32 {d28, d29}, [r0]!\n"
1152        "vld1.32 {d14, d15}, [r0]!\n"
1153        "vld1.32 {d22, d23}, [r0]!\n"
1154        "vld1.32 {d30, d31}, [r0]!\n"
1155
1156  #define NEON_32BIT_ROTATING_FLOAT_KERNEL_TRANSPOSE_ACCUMULATOR_CELLS \
1157        "vtrn.32 q4, q5\n" \
1158        "vtrn.32 q6, q7\n" \
1159        "vswp d9, d12\n" \
1160        "vswp d11, d14\n" \
1161        "vtrn.32 q8, q9\n" \
1162        "vtrn.32 q10, q11\n" \
1163        "vswp d17, d20\n" \
1164        "vswp d19, d22\n" \
1165        "vtrn.32 q12, q13\n" \
1166        "vtrn.32 q14, q15\n" \
1167        "vswp d25, d28\n" \
1168        "vswp d27, d30\n"
1169
1170  #define NEON_32BIT_ROTATING_FLOAT_KERNEL_ROTATE_ACCUMULATOR_CELLS(a, b, c) \
```

```
1171        NEON_32BIT_ROTATING_FLOAT_KERNEL_TRANSPOSE_ACCUMULATOR_CELLS \
1172        "vext.32 q5, q5, q5, #" #a "\n" \
1173        "vext.32 q6, q6, q6, #" #b "\n" \
1174        "vext.32 q7, q7, q7, #" #c "\n" \
1175        "vext.32 q9, q9, q9, #" #a "\n" \
1176        "vext.32 q10, q10, q10, #" #b "\n" \
1177        "vext.32 q11, q11, q11, #" #c "\n" \
1178        "vext.32 q13, q13, q13, #" #a "\n" \
1179        "vext.32 q14, q14, q14, #" #b "\n" \
1180        "vext.32 q15, q15, q15, #" #c "\n" \
1181        NEON_32BIT_ROTATING_FLOAT_KERNEL_TRANSPOSE_ACCUMULATOR_CELLS
1182
1183        NEON_32BIT_ROTATING_FLOAT_KERNEL_ROTATE_ACCUMULATOR_CELLS(1, 2, 3)
1184
1185        "loop_%=:\n"
1186
1187        // Load 1 Rhs cell of size 1x4
1188        "vld1.32 {d0, d1}, [%[rhs_ptr]]!\n"
1189
1190        // Load 3 Lhs cells of size 4x1 each
1191        "vld1.32 {d2, d3}, [%[lhs_ptr]]!\n"
1192        "vld1.32 {d4, d5}, [%[lhs_ptr]]!\n"
1193        "vld1.32 {d6, d7}, [%[lhs_ptr]]!\n"
1194
1195        // Multiply-accumulate
1196        "vmla.f32 q4, q1, q0\n"
1197        "vmla.f32 q8, q2, q0\n"
1198        "vmla.f32 q12, q3, q0\n"
1199        "vext.f32 q0, q0, q0, #1\n"
1200        "vmla.f32 q5, q1, q0\n"
1201        "vmla.f32 q9, q2, q0\n"
1202        "vmla.f32 q13, q3, q0\n"
1203        "vext.f32 q0, q0, q0, #1\n"
1204        "vmla.f32 q6, q1, q0\n"
1205        "vmla.f32 q10, q2, q0\n"
1206        "vmla.f32 q14, q3, q0\n"
```

```
1207        "vext.f32 q0, q0, q0, #1\n"
1208        "vmla.f32 q7, q1, q0\n"
1209        "vmla.f32 q11, q2, q0\n"
1210        "vmla.f32 q15, q3, q0\n"
1211
1212        // Loop. Decrement loop index (depth) by 1, since we just handled 1
1213        // level of depth.
1214        "subs %[depth], #1\n"
1215        "bne loop_%=\n"
1216
1217        // Store accumulators
1218        "mov r0, %[accum_ptr]\n"
1219
1220        NEON_32BIT_ROTATING_FLOAT_KERNEL_ROTATE_ACCUMULATOR_CELLS(3, 2, 1)
1221
1222        "vst1.32 {d8, d9},   [r0]!\n"
1223        "vst1.32 {d16, d17}, [r0]!\n"
1224        "vst1.32 {d24, d25}, [r0]!\n"
1225        "vst1.32 {d10, d11}, [r0]!\n"
1226        "vst1.32 {d18, d19}, [r0]!\n"
1227        "vst1.32 {d26, d27}, [r0]!\n"
1228        "vst1.32 {d12, d13}, [r0]!\n"
1229        "vst1.32 {d20, d21}, [r0]!\n"
1230        "vst1.32 {d28, d29}, [r0]!\n"
1231        "vst1.32 {d14, d15}, [r0]!\n"
1232        "vst1.32 {d22, d23}, [r0]!\n"
1233        "vst1.32 {d30, d31}, [r0]!\n"
1234        :  // outputs
1235        [lhs_ptr] "+r"(lhs_ptr), [rhs_ptr] "+r"(rhs_ptr),
1236        [depth] "+r"(depth)
1237        :  // inputs
1238        [accum_ptr] "r"(accum_ptr)
1239        :  // clobbers
1240        "cc", "memory", "r0",
1241        "d0", "d1", "d2", "d3", "d4", "d5", "d6", "d7", "d8", "d9", "d10",
1242        "d11", "d12", "d13", "d14", "d15", "d16", "d17", "d18", "d19", "d20",
```

```
1243        "d21", "d22", "d23", "d24", "d25", "d26", "d27", "d28", "d29", "d30",
1244        "d31");
1245    }
1246  };
1247
1248  // This rotating variant performs well when permutations (vext) can be dual-issued
1249  // with arithmetic instructions.
1250  // It is relevant as the rotating approach removes the need for multiply-with-scalar
1251  // instructions, and ARMv7 FMA does not have a with-scalar variant.
1252  struct NEON_32bit_GEMM_Float32_FMA_Rotating {
1253    typedef float OperandType;
1254    typedef float AccumulatorType;
1255    typedef KernelFormat<KernelSideFormat<CellFormat<4, 1, CellOrder::DepthMajor>, 3>,
1256                         KernelSideFormat<CellFormat<4, 1, CellOrder::DepthMajor>, 1> >
1257        Format;
1258    static void Run(const OperandType* lhs_ptr, const OperandType* rhs_ptr, AccumulatorType* accum_ptr, int depth) {
1259      asm volatile(
1260        // Load accumulators
1261        "mov r0, %[accum_ptr]\n"
1262        "vld1.32 {d8, d9},   [r0]!\n"
1263        "vld1.32 {d16, d17}, [r0]!\n"
1264        "vld1.32 {d24, d25}, [r0]!\n"
1265        "vld1.32 {d10, d11}, [r0]!\n"
1266        "vld1.32 {d18, d19}, [r0]!\n"
1267        "vld1.32 {d26, d27}, [r0]!\n"
1268        "vld1.32 {d12, d13}, [r0]!\n"
1269        "vld1.32 {d20, d21}, [r0]!\n"
1270        "vld1.32 {d28, d29}, [r0]!\n"
1271        "vld1.32 {d14, d15}, [r0]!\n"
1272        "vld1.32 {d22, d23}, [r0]!\n"
1273        "vld1.32 {d30, d31}, [r0]!\n"
1274
1275        NEON_32BIT_ROTATING_FLOAT_KERNEL_ROTATE_ACCUMULATOR_CELLS(1, 2, 3)
1276
1277        "loop_%=:\n"
1278
```

```
1279        // Load 1 Rhs cell of size 1x4
1280        "vld1.32 {d0, d1}, [%[rhs_ptr]]!\n"
1281
1282        // Load 3 Lhs cells of size 4x1 each
1283        "vld1.32 {d2, d3}, [%[lhs_ptr]]!\n"
1284        "vld1.32 {d4, d5}, [%[lhs_ptr]]!\n"
1285        "vld1.32 {d6, d7}, [%[lhs_ptr]]!\n"
1286
1287        // Multiply-accumulate
1288        "vfma.f32 q4, q1, q0\n"
1289        "vfma.f32 q8, q2, q0\n"
1290        "vfma.f32 q12, q3, q0\n"
1291        "vext.f32 q0, q0, q0, #1\n"
1292        "vfma.f32 q5, q1, q0\n"
1293        "vfma.f32 q9, q2, q0\n"
1294        "vfma.f32 q13, q3, q0\n"
1295        "vext.f32 q0, q0, q0, #1\n"
1296        "vfma.f32 q6, q1, q0\n"
1297        "vfma.f32 q10, q2, q0\n"
1298        "vfma.f32 q14, q3, q0\n"
1299        "vext.f32 q0, q0, q0, #1\n"
1300        "vfma.f32 q7, q1, q0\n"
1301        "vfma.f32 q11, q2, q0\n"
1302        "vfma.f32 q15, q3, q0\n"
1303
1304        // Loop. Decrement loop index (depth) by 1, since we just handled 1
1305        // level of depth.
1306        "subs %[depth], #1\n"
1307        "bne loop_%=\n"
1308
1309        NEON_32BIT_ROTATING_FLOAT_KERNEL_ROTATE_ACCUMULATOR_CELLS(3, 2, 1)
1310
1311        // Store accumulators
1312        "mov r0, %[accum_ptr]\n"
1313        "vst1.32 {d8, d9},   [r0]!\n"
1314        "vst1.32 {d16, d17}, [r0]!\n"
```

```
1315        "vst1.32 {d24, d25}, [r0]!\n"
1316        "vst1.32 {d10, d11}, [r0]!\n"
1317        "vst1.32 {d18, d19}, [r0]!\n"
1318        "vst1.32 {d26, d27}, [r0]!\n"
1319        "vst1.32 {d12, d13}, [r0]!\n"
1320        "vst1.32 {d20, d21}, [r0]!\n"
1321        "vst1.32 {d28, d29}, [r0]!\n"
1322        "vst1.32 {d14, d15}, [r0]!\n"
1323        "vst1.32 {d22, d23}, [r0]!\n"
1324        "vst1.32 {d30, d31}, [r0]!\n"
1325        :  // outputs
1326        [lhs_ptr] "+r"(lhs_ptr), [rhs_ptr] "+r"(rhs_ptr),
1327        [depth] "+r"(depth)
1328        :  // inputs
1329        [accum_ptr] "r"(accum_ptr)
1330        :  // clobbers
1331        "cc", "memory", "r0",
1332        "d0", "d1", "d2", "d3", "d4", "d5", "d6", "d7", "d8", "d9", "d10",
1333        "d11", "d12", "d13", "d14", "d15", "d16", "d17", "d18", "d19", "d20",
1334        "d21", "d22", "d23", "d24", "d25", "d26", "d27", "d28", "d29", "d30",
1335        "d31");
1336    }
1337  };
1338
1339  #endif  // __arm__
1340
1341  #ifdef __aarch64__
1342
1343  // This is the current standard kernel in gemmlowp, see:
1344  // https://github.com/google/gemmlowp/blob/b1e2a29ff866680028f3080efc244e10e8dd7f46/internal/kernel_neon.h#L646
1345  struct NEON_64bit_GEMM_Uint8Operands_Uint32Accumulators {
1346    typedef std::uint8_t OperandType;
1347    typedef std::uint32_t AccumulatorType;
1348    typedef KernelFormat<KernelSideFormat<CellFormat<4, 2, CellOrder::DepthMajor>, 3>,
1349                         KernelSideFormat<CellFormat<4, 2, CellOrder::DepthMajor>, 2> >
1350        Format;
```

```
1351    static void Run(const OperandType* lhs_ptr, const OperandType* rhs_ptr, AccumulatorType* accum_ptr, int depth) {
1352      asm volatile(
1353        // Load accumulators
1354        "mov x0, %[accum_ptr]\n"
1355        "ld1 {v8.16b}, [x0], #16\n"
1356        "ld1 {v16.16b}, [x0], #16\n"
1357        "ld1 {v24.16b}, [x0], #16\n"
1358        "ld1 {v9.16b}, [x0], #16\n"
1359        "ld1 {v17.16b}, [x0], #16\n"
1360        "ld1 {v25.16b}, [x0], #16\n"
1361        "ld1 {v10.16b}, [x0], #16\n"
1362        "ld1 {v18.16b}, [x0], #16\n"
1363        "ld1 {v26.16b}, [x0], #16\n"
1364        "ld1 {v11.16b}, [x0], #16\n"
1365        "ld1 {v19.16b}, [x0], #16\n"
1366        "ld1 {v27.16b}, [x0], #16\n"
1367        "ld1 {v12.16b}, [x0], #16\n"
1368        "ld1 {v20.16b}, [x0], #16\n"
1369        "ld1 {v28.16b}, [x0], #16\n"
1370        "ld1 {v13.16b}, [x0], #16\n"
1371        "ld1 {v21.16b}, [x0], #16\n"
1372        "ld1 {v29.16b}, [x0], #16\n"
1373        "ld1 {v14.16b}, [x0], #16\n"
1374        "ld1 {v22.16b}, [x0], #16\n"
1375        "ld1 {v30.16b}, [x0], #16\n"
1376        "ld1 {v15.16b}, [x0], #16\n"
1377        "ld1 {v23.16b}, [x0], #16\n"
1378        "ld1 {v31.16b}, [x0], #16\n"
1379
1380        "loop_%=:\n"
1381
1382        // Overview of register layout:
1383        //
1384        // A 2x8 block of 2 2x4 cells of Rhs is stored in 16bit in v0--v1.
1385        // A 12x2 block of 3 4x2 cells Lhs is stored in 16bit in v2--v4.
1386        // A 12x8 block of accumulators is stored in 32bit in v8--v31.
```

```
1387        //
1388        //                          +--------+--------+-----+--------+--------+
1389        //                          |v0.h[0] |v0.h[1] | ... |v1.h[2] |v1.h[3] |
1390        //                    Rhs   +--------+--------+-----+--------+--------+
1391        //                          |v0.h[4] |v0.h[5] | ... |v1.h[6] |v1.h[7] |
1392        //                          +--------+--------+-----+--------+--------+
1393        //
1394        //                          |        |        |     |        |        |
1395        //
1396        //    Lhs                   |        |        |     |        |        |
1397        //
1398        //  +-------+-------+ - -   +--------+--------+-----+--------+--------+
1399        //  |v2.h[0]|v2.h[4]|       |v8.s[0] |v9.s[0] | ... |v14.s[0]|v15.s[0]|
1400        //  |v2.h[1]|v2.h[5]|       |v8.s[1] |v9.s[1] | ... |v14.s[1]|v15.s[1]|
1401        //  |v2.h[2]|v2.h[6]|       |v8.s[2] |v9.s[2] | ... |v14.s[2]|v15.s[2]|
1402        //  |v2.h[3]|v2.h[7]|       |v8.s[3] |v9.s[3] | ... |v14.s[3]|v15.s[3]|
1403        //  +-------+-------+ - -   +--------+--------+-----+--------+--------+
1404        //  |v3.h[0]|v3.h[4]|       |v16.s[0]|v17.s[0]| ... |v22.s[0]|v23.s[0]|
1405        //  |v3.h[1]|v3.h[5]|       |v16.s[1]|v17.s[1]| ... |v22.s[1]|v23.s[1]|
1406        //  |v3.h[2]|v3.h[6]|       |v16.s[2]|v17.s[2]| ... |v22.s[2]|v23.s[2]|
1407        //  |v3.h[3]|v3.h[7]|       |v16.s[3]|v17.s[3]| ... |v22.s[3]|v23.s[3]|
1408        //  +-------+-------+ - -   +--------+--------+-----+--------+--------+
1409        //  |v4.h[0]|v4.h[4]|       |v24.s[0]|v25.s[0]| ... |v30.s[0]|v31.s[0]|
1410        //  |v4.h[1]|v4.h[5]|       |v24.s[1]|v25.s[1]| ... |v30.s[1]|v31.s[1]|
1411        //  |v4.h[2]|v4.h[6]|       |v24.s[2]|v25.s[2]| ... |v30.s[2]|v31.s[2]|
1412        //  |v4.h[3]|v4.h[7]|       |v24.s[3]|v25.s[3]| ... |v30.s[3]|v31.s[3]|
1413        //  +-------+-------+ - -   +--------+--------+-----+--------+--------+
1414        //
1415        //                             Accumulator
1416
1417        // Load 1 Rhs cell of size 2x8
1418        "ld1 {v0.8b}, [%[rhs_ptr]], #8\n"
1419        "ld1 {v1.8b}, [%[rhs_ptr]], #8\n"
1420
1421        // Load 3 Lhs cells of size 4x2 each
1422        "ld1 {v2.8b}, [%[lhs_ptr]], #8\n"
```

```
1423        "ld1 {v3.8b}, [%[lhs_ptr]], #8\n"
1424        "ld1 {v4.8b}, [%[lhs_ptr]], #8\n"
1425
1426        // Expand Lhs/Rhs cells to 16 bit.
1427        "uxtl v0.8h, v0.8b\n"
1428        "uxtl v1.8h, v1.8b\n"
1429        "uxtl v2.8h, v2.8b\n"
1430        "uxtl v3.8h, v3.8b\n"
1431        "uxtl v4.8h, v4.8b\n"
1432
1433        // Multiply-accumulate, level of depth 0
1434        "umlal v8.4s, v2.4h, v0.h[0]\n"
1435        "umlal v9.4s, v2.4h, v0.h[1]\n"
1436        "umlal v10.4s, v2.4h, v0.h[2]\n"
1437        "umlal v11.4s, v2.4h, v0.h[3]\n"
1438        "umlal v12.4s, v2.4h, v1.h[0]\n"
1439        "umlal v13.4s, v2.4h, v1.h[1]\n"
1440        "umlal v14.4s, v2.4h, v1.h[2]\n"
1441        "umlal v15.4s, v2.4h, v1.h[3]\n"
1442        "umlal v16.4s, v3.4h, v0.h[0]\n"
1443        "umlal v17.4s, v3.4h, v0.h[1]\n"
1444        "umlal v18.4s, v3.4h, v0.h[2]\n"
1445        "umlal v19.4s, v3.4h, v0.h[3]\n"
1446        "umlal v20.4s, v3.4h, v1.h[0]\n"
1447        "umlal v21.4s, v3.4h, v1.h[1]\n"
1448        "umlal v22.4s, v3.4h, v1.h[2]\n"
1449        "umlal v23.4s, v3.4h, v1.h[3]\n"
1450        "umlal v24.4s, v4.4h, v0.h[0]\n"
1451        "umlal v25.4s, v4.4h, v0.h[1]\n"
1452        "umlal v26.4s, v4.4h, v0.h[2]\n"
1453        "umlal v27.4s, v4.4h, v0.h[3]\n"
1454        "umlal v28.4s, v4.4h, v1.h[0]\n"
1455        "umlal v29.4s, v4.4h, v1.h[1]\n"
1456        "umlal v30.4s, v4.4h, v1.h[2]\n"
1457        "umlal v31.4s, v4.4h, v1.h[3]\n"
1458
```

```
1459        // Multiply-accumulate, level of depth 1
1460        "umlal2 v8.4s, v2.8h, v0.h[4]\n"
1461        "umlal2 v9.4s, v2.8h, v0.h[5]\n"
1462        "umlal2 v10.4s, v2.8h, v0.h[6]\n"
1463        "umlal2 v11.4s, v2.8h, v0.h[7]\n"
1464        "umlal2 v12.4s, v2.8h, v1.h[4]\n"
1465        "umlal2 v13.4s, v2.8h, v1.h[5]\n"
1466        "umlal2 v14.4s, v2.8h, v1.h[6]\n"
1467        "umlal2 v15.4s, v2.8h, v1.h[7]\n"
1468        "umlal2 v16.4s, v3.8h, v0.h[4]\n"
1469        "umlal2 v17.4s, v3.8h, v0.h[5]\n"
1470        "umlal2 v18.4s, v3.8h, v0.h[6]\n"
1471        "umlal2 v19.4s, v3.8h, v0.h[7]\n"
1472        "umlal2 v20.4s, v3.8h, v1.h[4]\n"
1473        "umlal2 v21.4s, v3.8h, v1.h[5]\n"
1474        "umlal2 v22.4s, v3.8h, v1.h[6]\n"
1475        "umlal2 v23.4s, v3.8h, v1.h[7]\n"
1476        "umlal2 v24.4s, v4.8h, v0.h[4]\n"
1477        "umlal2 v25.4s, v4.8h, v0.h[5]\n"
1478        "umlal2 v26.4s, v4.8h, v0.h[6]\n"
1479        "umlal2 v27.4s, v4.8h, v0.h[7]\n"
1480        "umlal2 v28.4s, v4.8h, v1.h[4]\n"
1481        "umlal2 v29.4s, v4.8h, v1.h[5]\n"
1482        "umlal2 v30.4s, v4.8h, v1.h[6]\n"
1483        "umlal2 v31.4s, v4.8h, v1.h[7]\n"
1484
1485        // Loop. Decrement loop index (depth) by 2, since we just handled 2
1486        // levels of depth.
1487        "subs %w[depth], %w[depth], #2\n"
1488        "bne loop_%=\n"
1489
1490        // Store accumulators
1491        "mov x0, %[accum_ptr]\n"
1492        "st1 {v8.16b}, [x0], #16\n"
1493        "st1 {v16.16b}, [x0], #16\n"
1494        "st1 {v24.16b}, [x0], #16\n"
```

```
1495        "st1 {v9.16b}, [x0], #16\n"
1496        "st1 {v17.16b}, [x0], #16\n"
1497        "st1 {v25.16b}, [x0], #16\n"
1498        "st1 {v10.16b}, [x0], #16\n"
1499        "st1 {v18.16b}, [x0], #16\n"
1500        "st1 {v26.16b}, [x0], #16\n"
1501        "st1 {v11.16b}, [x0], #16\n"
1502        "st1 {v19.16b}, [x0], #16\n"
1503        "st1 {v27.16b}, [x0], #16\n"
1504        "st1 {v12.16b}, [x0], #16\n"
1505        "st1 {v20.16b}, [x0], #16\n"
1506        "st1 {v28.16b}, [x0], #16\n"
1507        "st1 {v13.16b}, [x0], #16\n"
1508        "st1 {v21.16b}, [x0], #16\n"
1509        "st1 {v29.16b}, [x0], #16\n"
1510        "st1 {v14.16b}, [x0], #16\n"
1511        "st1 {v22.16b}, [x0], #16\n"
1512        "st1 {v30.16b}, [x0], #16\n"
1513        "st1 {v15.16b}, [x0], #16\n"
1514        "st1 {v23.16b}, [x0], #16\n"
1515        "st1 {v31.16b}, [x0], #16\n"
1516        :  // outputs
1517        [lhs_ptr] "+r"(lhs_ptr), [rhs_ptr] "+r"(rhs_ptr),
1518        [depth] "+r"(depth)
1519        :  // inputs
1520        [accum_ptr] "r"(accum_ptr)
1521        :  // clobbers
1522        "cc", "memory", "x0", "v0", "v1", "v2", "v3", "v4", "v5", "v6",
1523        "v7", "v8", "v9", "v10", "v11", "v12", "v13", "v14", "v15", "v16",
1524        "v17", "v18", "v19", "v20", "v21", "v22", "v23", "v24", "v25", "v26",
1525        "v27", "v28", "v29", "v30", "v31");
1526    }
1527  };
1528
1529
1530  // Faster kernel by ARM. Not expanding operands before multiplication.
```

```
1531  // Tuned for A57. Compare to NEON_32bit_GEMM_Uint8Operands_Uint32Accumulators_noexpand
1532  struct NEON_64bit_GEMM_Uint8Operands_Uint32Accumulators_noexpand_A57 {
1533    typedef std::uint8_t OperandType;
1534    typedef std::uint32_t AccumulatorType;
1535    typedef KernelFormat<KernelSideFormat<CellFormat<5, 16, CellOrder::WidthMajor>, 1>,
1536                         KernelSideFormat<CellFormat<4, 16, CellOrder::WidthMajor>, 1> >
1537        Format;
1538    static void Run(const OperandType* lhs_ptr, const OperandType* rhs_ptr, AccumulatorType* accum_ptr, int depth) {
1539      static const int kLhsWidth = Format::Lhs::kWidth;
1540      static const int kRhsWidth = Format::Rhs::kWidth;
1541      AccumulatorType rowmajor_accumulator_buffer[kLhsWidth * kRhsWidth];
1542      asm volatile(
1543        // Clear aggregators
1544        "dup v12.4s, wzr\n"
1545        "dup v13.4s, wzr\n"
1546        "dup v14.4s, wzr\n"
1547        "dup v15.4s, wzr\n"
1548        "dup v16.4s, wzr\n"
1549        "dup v17.4s, wzr\n"
1550        "dup v18.4s, wzr\n"
1551        "dup v19.4s, wzr\n"
1552        "dup v20.4s, wzr\n"
1553        "dup v21.4s, wzr\n"
1554        "dup v22.4s, wzr\n"
1555        "dup v23.4s, wzr\n"
1556        "dup v24.4s, wzr\n"
1557        "dup v25.4s, wzr\n"
1558        "dup v26.4s, wzr\n"
1559        "dup v27.4s, wzr\n"
1560        "dup v28.4s, wzr\n"
1561        "dup v29.4s, wzr\n"
1562        "dup v30.4s, wzr\n"
1563        "dup v31.4s, wzr\n"
1564
1565        "loop_%=:\n"
1566
```

```
1567        // Overview of register layout:
1568        //
1569        // A 4x16 block of Rhs is stored in 8 bit in v0--v3.
1570        // A 5x16 block of Lhs is cycled through v4 and v5 in 8 bit.
1571        //
1572        // A 4x5 block of aggregators is stored in v12-v31 (as 4x32 bit
1573        // components which would need to be added at the end)
1574        //
1575        // The Lhs vectors are multiplied by the Rhs vectors with a widening
1576        // multiply to produce an intermediate result which is stored in
1577        // v6-v11.  Each intermediate result is 8x16 bits so this happens
1578        // twice for each Lhs/Rhs combination (once with UMULL for elements
1579        // 0-7 and once with UMULL2 for elements 8-15).
1580        //
1581        // UADALP is used to accumulate these intermediate results into the
1582        // result aggregators.
1583        //
1584        //
1585        //
1586        //                              +--------+--------+--------+--------+
1587        //                              |v0.b[0] |v1.b[0] |v2.b[0] |v3.b[0] |
1588        //                         Rhs  +--------+--------+--------+--------+
1589        //                              |  ...   |  ...   |  ...   |  ...   |
1590        //                              +--------+--------+--------+--------|
1591        //                              |v0.b[15]|v1.b[15]|v2.b[15]|v3.b[15]|
1592        //                              +--------+--------+--------+--------+
1593        //
1594        //                              |        |        |        |        |
1595        //
1596        //    Lhs                       |        |        |        |        |
1597        //
1598        //  +-------+-----+--------+ - - +--------+--------+--------+--------+
1599        //  |v4.b[0]| ... |v4.b[15]|     | v12.4s | v13.4s | v14.4s | v15.4s |
1600        //  |v5.b[0]| ... |v5.b[15]|     | v16.4s | v17.4s | v18.4s | v19.4s |
1601        //  |v4.b[0]| ... |v4.b[15]|     | v20.4s | v21.4s | v22.4s | v23.4s |
1602        //  |v5.b[0]| ... |v5.b[15]|     | v24.4s | v25.4s | v26.4s | v27.4s |
```

```
1603        //  |v4.b[0]| ... |v4.b[15]|     | v28.4s | v29.4s | v30.4s | v31.4s |
1604        //  +-------+--------------+ - - +--------+--------+--------+--------+
1605        //
1606        //                                              Accumulator
1607        //
1608        //
1609        // Further possible optimisations (not tried):
1610        //   - Move early loads into previous iteration (see Float32_WithScalar for example).
1611        //   - Unroll loop 2x to alternate more smoothly between v4 and v5.
1612        //   - A different number of temporary registers might work better.
1613        //   - Pairing umull with corresponding umull2 might allow better
1614        //     register loading (e.g. at the start of the loop)
1615        //   - Interleaving umull{2} and uadalp even more aggressively might
1616        //     help, (not sure about latency vs. dispatch rate).
1617        //
1618        //
1619        // Start loading Rhs - further loads are interleaved amongst the
1620        // multiplies for better dispatch on A57.
1621        "ld1 {v0.16b}, [%[rhs_ptr]], #16\n"
1622
1623        // Load first Lhs vector - further loads are interleaved amongst the multiplies
1624        "ld1 {v4.16b}, [%[lhs_ptr]], #16\n"
1625
1626        "umull    v6.8h,  v0.8b,  v4.8b\n"
1627        "ld1 {v1.16b}, [%[rhs_ptr]], #16\n" // 2nd RHS element
1628        "umull    v7.8h,  v1.8b,  v4.8b\n"
1629        "ld1 {v2.16b}, [%[rhs_ptr]], #16\n" // 3rd RHS element
1630        "umull    v8.8h,  v2.8b,  v4.8b\n"
1631        "ld1 {v3.16b}, [%[rhs_ptr]], #16\n" // 4th RHS element
1632        "umull    v9.8h,  v3.8b,  v4.8b\n"
1633        "umull2  v10.8h, v0.16b, v4.16b\n"
1634        "umull2  v11.8h, v1.16b, v4.16b\n"
1635        "ld1 {v5.16b}, [%[lhs_ptr]], #16\n" // 2nd LHS element
1636
1637        "uadalp  v12.4s, v6.8h\n"
1638        "umull2   v6.8h, v2.16b, v4.16b\n"
```

```
1639        "uadalp  v13.4s, v7.8h\n"
1640        "umull2   v7.8h, v3.16b, v4.16b\n"
1641        "ld1 {v4.16b}, [%[lhs_ptr]], #16\n" // 1st LHS element done - Reuse v4 for 3rd LHS element
1642        "uadalp  v14.4s, v8.8h\n"
1643        "umull    v8.8h,  v0.8b,  v5.8b\n"
1644        "uadalp  v15.4s, v9.8h\n"
1645        "umull    v9.8h,  v1.8b,  v5.8b\n"
1646        "uadalp  v12.4s, v10.8h\n"
1647        "umull   v10.8h,  v2.8b,  v5.8b\n"
1648        "uadalp  v13.4s, v11.8h\n"
1649        "umull   v11.8h,  v3.8b,  v5.8b\n"
1650
1651        "uadalp  v14.4s, v6.8h\n"
1652        "umull2   v6.8h, v0.16b, v5.16b\n"
1653        "uadalp  v15.4s, v7.8h\n"
1654        "umull2   v7.8h, v1.16b, v5.16b\n"
1655        "uadalp  v16.4s, v8.8h\n"
1656        "umull2   v8.8h, v2.16b, v5.16b\n"
1657        "uadalp  v17.4s, v9.8h\n"
1658        "umull2   v9.8h, v3.16b, v5.16b\n"
1659        "ld1 {v5.16b}, [%[lhs_ptr]], #16\n" // 2nd LHS element done - Reuse v5 for 4th LHS element
1660        "uadalp  v18.4s, v10.8h\n"
1661        "umull   v10.8h,  v0.8b,  v4.8b\n"
1662        "uadalp  v19.4s, v11.8h\n"
1663        "umull   v11.8h,  v1.8b,  v4.8b\n"
1664
1665        "uadalp  v16.4s, v6.8h\n"
1666        "umull    v6.8h,  v2.8b,  v4.8b\n"
1667        "uadalp  v17.4s, v7.8h\n"
1668        "umull    v7.8h,  v3.8b,  v4.8b\n"
1669        "uadalp  v18.4s, v8.8h\n"
1670        "umull2   v8.8h, v0.16b, v4.16b\n"
1671        "uadalp  v19.4s, v9.8h\n"
1672        "umull2   v9.8h, v1.16b, v4.16b\n"
1673        "uadalp  v20.4s, v10.8h\n"
1674        "umull2  v10.8h, v2.16b, v4.16b\n"
```

```
1675        "uadalp  v21.4s, v11.8h\n"
1676        "umull2  v11.8h, v3.16b, v4.16b\n"
1677        "ld1 {v4.16b}, [%[lhs_ptr]], #16\n" // 3rd LHS element done - Reuse v4 for 5th LHS element
1678
1679        "uadalp v22.4s, v6.8h\n"
1680        "umull    v6.8h,  v0.8b,  v5.8b\n"
1681        "uadalp v23.4s, v7.8h\n"
1682        "umull    v7.8h,  v1.8b,  v5.8b\n"
1683        "uadalp v20.4s, v8.8h\n"
1684        "umull    v8.8h,  v2.8b,  v5.8b\n"
1685        "uadalp v21.4s, v9.8h\n"
1686        "umull    v9.8h,  v3.8b,  v5.8b\n"
1687        "uadalp v22.4s, v10.8h\n"
1688        "umull2  v10.8h, v0.16b, v5.16b\n"
1689        "uadalp v23.4s, v11.8h\n"
1690        "umull2  v11.8h, v1.16b, v5.16b\n"
1691
1692        "uadalp v24.4s, v6.8h\n"
1693        "umull2   v6.8h,  v2.16b, v5.16b\n"
1694        "uadalp v25.4s, v7.8h\n"
1695        "umull2   v7.8h,  v3.16b, v5.16b\n"
1696        "uadalp v26.4s, v8.8h\n"
1697        "umull    v8.8h,  v0.8b,  v4.8b\n"
1698        "uadalp v27.4s, v9.8h\n"
1699        "umull    v9.8h,  v1.8b,  v4.8b\n"
1700        "uadalp v24.4s, v10.8h\n"
1701        "umull   v10.8h,  v2.8b,  v4.8b\n"
1702        "uadalp v25.4s, v11.8h\n"
1703        "umull   v11.8h,  v3.8b,  v4.8b\n"
1704
1705        "uadalp v26.4s, v6.8h\n"
1706        "umull2   v6.8h, v0.16b, v4.16b\n"
1707        "uadalp v27.4s, v7.8h\n"
1708        "umull2   v7.8h, v1.16b, v4.16b\n"
1709        "uadalp v28.4s, v8.8h\n"
1710        "umull2   v8.8h, v2.16b, v4.16b\n"
```

```
1711        "uadalp v29.4s, v9.8h\n"
1712        "umull2   v9.8h, v3.16b, v4.16b\n"
1713        "uadalp v30.4s, v10.8h\n"
1714        "uadalp v31.4s, v11.8h\n"
1715
1716
1717        "uadalp v28.4s, v6.8h\n"
1718        "uadalp v29.4s, v7.8h\n"
1719        // Loop. Decrement loop index (depth) by 16, since we just handled
1720        // 16 levels of depth.  Do this subs a bit before the end of the loop
1721        // for better dispatch on A57.
1722        "subs %w[depth], %w[depth], #16\n"
1723        "uadalp v30.4s, v8.8h\n"
1724        "uadalp v31.4s, v9.8h\n"
1725
1726        "bne loop_%=\n"
1727
1728        // Reduce aggregators horizontally
1729        "addp v0.4s, v12.4s, v13.4s\n"
1730        "addp v1.4s, v14.4s, v15.4s\n"
1731        "addp v2.4s, v16.4s, v17.4s\n"
1732        "addp v3.4s, v18.4s, v19.4s\n"
1733        "addp v4.4s, v20.4s, v21.4s\n"
1734        "addp v5.4s, v22.4s, v23.4s\n"
1735        "addp v6.4s, v24.4s, v25.4s\n"
1736        "addp v7.4s, v26.4s, v27.4s\n"
1737        "addp v8.4s, v28.4s, v29.4s\n"
1738        "addp v9.4s, v30.4s, v31.4s\n"
1739
1740        "addp v10.4s, v0.4s, v1.4s\n"
1741        "addp v11.4s, v2.4s, v3.4s\n"
1742        "addp v12.4s, v4.4s, v5.4s\n"
1743        "addp v13.4s, v6.4s, v7.4s\n"
1744        "addp v14.4s, v8.4s, v9.4s\n"
1745
1746        "mov x0, %[rowmajor_accumulator_buffer]\n"
```

```
1747        "st1 {v10.16b}, [x0], #16\n"
1748        "st1 {v11.16b}, [x0], #16\n"
1749        "st1 {v12.16b}, [x0], #16\n"
1750        "st1 {v13.16b}, [x0], #16\n"
1751        "st1 {v14.16b}, [x0], #16\n"
1752        :  // outputs
1753        [lhs_ptr] "+r"(lhs_ptr), [rhs_ptr] "+r"(rhs_ptr),
1754        [depth] "+r"(depth)
1755        :  // inputs
1756        [rowmajor_accumulator_buffer] "r"(rowmajor_accumulator_buffer)
1757        :  // clobbers
1758        "cc", "memory", "x0", "v0", "v1", "v2", "v3", "v4", "v5", "v6",
1759        "v7", "v8", "v9", "v10", "v11", "v12", "v13", "v14", "v15", "v16",
1760        "v17", "v18", "v19", "v20", "v21", "v22", "v23", "v24", "v25", "v26",
1761        "v27", "v28", "v29", "v30", "v31");
1762
1763      // accumulate row-major accumulators into global (column-major) accumulators
1764      for (int l = 0; l < kLhsWidth; l++) {
1765        for (int r = 0; r < kRhsWidth; r++) {
1766          accum_ptr[l + kLhsWidth * r] +=
1767              rowmajor_accumulator_buffer[r + l * kRhsWidth];
1768        }
1769      }
1770    }
1771  };
1772
1773  // Fast kernel operating on int8 operands.
1774  // It is assumed that one of the two int8 operands only takes values
1775  // in [-127, 127], while the other may freely range in [-128, 127].
1776  // The issue with both operands taking the value -128 is that:
1777  // -128*-128 + -128*-128 == -32768 overflows int16.
1778  // Every other expression a*b + c*d, for any int8 a,b,c,d, fits in int16
1779  // range. That is the basic idea of this kernel.
1780  struct NEON_64bit_GEMM_Int8Operands_Int32Accumulators_AccumTwoWithin16Bits {
1781    typedef std::int8_t OperandType;
1782    typedef std::int32_t AccumulatorType;
```

```
1783    typedef KernelFormat<KernelSideFormat<CellFormat<4, 16, CellOrder::WidthMajor>, 1>,
1784                         KernelSideFormat<CellFormat<4, 16, CellOrder::WidthMajor>, 1> >
1785        Format;
1786    static void Run(const OperandType* lhs_ptr, const OperandType* rhs_ptr, AccumulatorType* accum_ptr, int depth) {
1787      static const int kLhsWidth = Format::Lhs::kWidth;
1788      static const int kRhsWidth = Format::Rhs::kWidth;
1789      AccumulatorType rowmajor_accumulator_buffer[kLhsWidth * kRhsWidth];
1790      asm volatile(
1791        // Clear accumulators
1792        "dup v16.4s, wzr\n"
1793        "dup v17.4s, wzr\n"
1794        "dup v18.4s, wzr\n"
1795        "dup v19.4s, wzr\n"
1796        "dup v20.4s, wzr\n"
1797        "dup v21.4s, wzr\n"
1798        "dup v22.4s, wzr\n"
1799        "dup v23.4s, wzr\n"
1800        "dup v24.4s, wzr\n"
1801        "dup v25.4s, wzr\n"
1802        "dup v26.4s, wzr\n"
1803        "dup v27.4s, wzr\n"
1804        "dup v28.4s, wzr\n"
1805        "dup v29.4s, wzr\n"
1806        "dup v30.4s, wzr\n"
1807        "dup v31.4s, wzr\n"
1808
1809        // Initial loads and arithmetic of the first loop iteration,
1810        // taken out of the loop so that in the loop itself we have
1811        // optimal streaming of data from memory.
1812        "ld1 {v0.16b}, [%[rhs_ptr]], #16\n"
1813        "ld1 {v4.16b}, [%[lhs_ptr]], #16\n"
1814        "ld1 {v1.16b}, [%[rhs_ptr]], #16\n"
1815        "ld1 {v5.16b}, [%[lhs_ptr]], #16\n"
1816        "ld1 {v2.16b}, [%[rhs_ptr]], #16\n"
1817        "ld1 {v3.16b}, [%[rhs_ptr]], #16\n"
1818
```

```
1819        "smull    v8.8h,  v0.8b,  v4.8b\n"
1820        "smull    v9.8h,  v1.8b,  v4.8b\n"
1821        "ld1 {v6.16b}, [%[lhs_ptr]], #16\n"
1822        "smull    v10.8h,  v2.8b,  v4.8b\n"
1823        "smull    v11.8h,  v3.8b,  v4.8b\n"
1824        "ld1 {v7.16b}, [%[lhs_ptr]], #16\n"
1825        "smull    v12.8h,  v0.8b,  v5.8b\n"
1826        "smull    v13.8h,  v1.8b,  v5.8b\n"
1827        "smull    v14.8h,  v2.8b,  v5.8b\n"
1828        "smull    v15.8h,  v3.8b,  v5.8b\n"
1829
1830        // Multiply-accumulate second-half, again into the same
1831        // 16bit local accumulator registers. This is where we
1832        // take advantage of having int8 instead of uint8 and therefore
1833        // being able to accumulate two products into int16.
1834        "smlal2   v8.8h,  v0.16b,  v4.16b\n"
1835        "smlal2   v9.8h,  v1.16b,  v4.16b\n"
1836        "smlal2   v10.8h,  v2.16b,  v4.16b\n"
1837        "smlal2   v11.8h,  v3.16b,  v4.16b\n"
1838        "smlal2   v12.8h,  v0.16b,  v5.16b\n"
1839        "smlal2   v13.8h,  v1.16b,  v5.16b\n"
1840        "smlal2   v14.8h,  v2.16b,  v5.16b\n"
1841        "smlal2   v15.8h,  v3.16b,  v5.16b\n"
1842
1843        "subs %w[depth], %w[depth], #16\n"
1844
1845        // If the loop depth is only 16, then we can skip the general loop
1846        // and go straight to the final part of the code.
1847        "beq after_loop_last16_%=\n"
1848
1849        // General loop.
1850        "loop_%=:\n"
1851
1852        // Overview of register layout:
1853        //
1854        // A 4x16 block of Rhs is stored in 8 bit in v0--v3.
```

```
1855        // A 4x16 block of Lhs is stored in 8 bit in v4--v7.
1856        //
1857        // A 4x4 block of accumulators is stored in v16-v31 (as 4x32 bit
1858        // components which need to be horizontally-added at the end)
1859        //
1860        // The Lhs vectors are multiplied by the Rhs vectors with a widening
1861        // multiply over the 8 first levels of depth, producing int16x8
1862        // vectors of products for each position in the accumulator matrix.
1863        // Here comes the special trick: since the operands are signed int8,
1864        // their range being [ -2^7 , 2^7 ), their products are in range
1865        // [ -2^14 , 2^14 - 1 ), meaning that we can add two such values
1866        // without any risk of overflowing int16.
1867        // We thus proceed with the 8 next levels of depth, multiplying
1868        // again Lhs by Rhs, accumulating into this existing int16x8 vector.
1869        //
1870        // Only then, having processed 16 levels of depth, do we need to
1871        // horizontally add these int16x8 accumulators into the final
1872        // int32x4 accumulators.
1873        //
1874        // As we do not have enough registers to store all 16 int16x8
1875        // temporary-16bit-accumulators, we have them cycle through v8--v15.
1876        //
1877        //
1878        // Register layout (ignoring the v8--v15 temporary 16bit accumulators):
1879        //
1880        //                               +--------+--------+--------+--------+
1881        //                               |v0.b[0] |v1.b[0] |v2.b[0] |v3.b[0] |
1882        //                          Rhs  +--------+--------+--------+--------+
1883        //                               |  ...   |  ...   |  ...   |  ...   |
1884        //                               +--------+--------+--------+--------|
1885        //                               |v0.b[15]|v1.b[15]|v2.b[15]|v3.b[15]|
1886        //                               +--------+--------+--------+--------+
1887        //
1888        //                               |        |        |        |        |
1889        //
1890        //    Lhs                        |        |        |        |        |
```

```
1891        //
1892        //  +-------+-----+--------+ - - +--------+--------+--------+--------+
1893        //  |v4.b[0]| ... |v4.b[15]|     | v16.4s | v17.4s | v18.4s | v19.4s |
1894        //  |v5.b[0]| ... |v5.b[15]|     | v20.4s | v21.4s | v22.4s | v23.4s |
1895        //  |v6.b[0]| ... |v6.b[15]|     | v24.4s | v25.4s | v26.4s | v27.4s |
1896        //  |v7.b[0]| ... |v7.b[15]|     | v28.4s | v29.4s | v30.4s | v31.4s |
1897        //  +-------+-----+--------+ - - +--------+--------+--------+--------+
1898        //
1899        //                                            Accumulator
1900        //
1901
1902        // Some multiplications and 16-bit accumulation were already done above,
1903        // so we start right away in the middle.
1904        "sadalp  v16.4s, v8.8h\n"
1905        "ld1 {v4.16b}, [%[lhs_ptr]], #16\n"
1906        "smull    v8.8h,  v0.8b,  v6.8b\n"
1907        "sadalp  v17.4s, v9.8h\n"
1908        "ld1 {v5.16b}, [%[lhs_ptr]], #16\n"
1909        "smull    v9.8h,  v1.8b,  v6.8b\n"
1910        "sadalp  v18.4s, v10.8h\n"
1911        "smull    v10.8h,  v2.8b,  v6.8b\n"
1912        "sadalp  v19.4s, v11.8h\n"
1913        "smull    v11.8h,  v3.8b,  v6.8b\n"
1914        "sadalp  v20.4s, v12.8h\n"
1915        "smull    v12.8h,  v0.8b,  v7.8b\n"
1916        "sadalp  v21.4s, v13.8h\n"
1917        "smull    v13.8h,  v1.8b,  v7.8b\n"
1918        "sadalp  v22.4s, v14.8h\n"
1919        "smull    v14.8h,  v2.8b,  v7.8b\n"
1920        "sadalp  v23.4s, v15.8h\n"
1921        "smull    v15.8h,  v3.8b,  v7.8b\n"
1922
1923        // Multiply-accumulate second-half, again into the same
1924        // 16bit local accumulator registers. This is where we
1925        // take advantage of having int8 instead of uint8 and therefore
1926        // being able to accumulate two products into int16.
```

```
1927        "smlal2   v8.8h,  v0.16b,  v6.16b\n"
1928        "smlal2   v9.8h,  v1.16b,  v6.16b\n"
1929        "smlal2   v10.8h,  v2.16b,  v6.16b\n"
1930        "smlal2   v11.8h,  v3.16b,  v6.16b\n"
1931
1932        "ld1 {v6.16b}, [%[lhs_ptr]], #16\n"
1933
1934        "smlal2   v12.8h,  v0.16b,  v7.16b\n"
1935        "ld1 {v0.16b}, [%[rhs_ptr]], #16\n"
1936        "smlal2   v13.8h,  v1.16b,  v7.16b\n"
1937        "ld1 {v1.16b}, [%[rhs_ptr]], #16\n"
1938        "smlal2   v14.8h,  v2.16b,  v7.16b\n"
1939        "ld1 {v2.16b}, [%[rhs_ptr]], #16\n"
1940        "smlal2   v15.8h,  v3.16b,  v7.16b\n"
1941        "ld1 {v3.16b}, [%[rhs_ptr]], #16\n"
1942
1943        "sadalp  v24.4s, v8.8h\n"
1944        "smull    v8.8h,  v0.8b,  v4.8b\n"
1945        "sadalp  v25.4s, v9.8h\n"
1946        "ld1 {v7.16b}, [%[lhs_ptr]], #16\n"
1947        "smull    v9.8h,  v1.8b,  v4.8b\n"
1948        "sadalp  v26.4s, v10.8h\n"
1949        "smull    v10.8h,  v2.8b,  v4.8b\n"
1950        "sadalp  v27.4s, v11.8h\n"
1951        "smull    v11.8h,  v3.8b,  v4.8b\n"
1952        "sadalp  v28.4s, v12.8h\n"
1953        "smull    v12.8h,  v0.8b,  v5.8b\n"
1954        "sadalp  v29.4s, v13.8h\n"
1955        "smull    v13.8h,  v1.8b,  v5.8b\n"
1956        "sadalp  v30.4s, v14.8h\n"
1957        "smull    v14.8h,  v2.8b,  v5.8b\n"
1958        "sadalp  v31.4s, v15.8h\n"
1959        "smull    v15.8h,  v3.8b,  v5.8b\n"
1960
1961        // Multiply-accumulate second-half, again into the same
1962        // 16bit local accumulator registers. This is where we
```

```
1963        // take advantage of having int8 instead of uint8 and therefore
1964        // being able to accumulate two products into int16.
1965        "smlal2   v8.8h,  v0.16b,  v4.16b\n"
1966        "smlal2   v9.8h,  v1.16b,  v4.16b\n"
1967        "smlal2   v10.8h,  v2.16b,  v4.16b\n"
1968        "smlal2   v11.8h,  v3.16b,  v4.16b\n"
1969
1970        // Loop. Decrement loop index (depth) by 16, since we just handled
1971        // 16 levels of depth.  Do this subs a bit before the end of the loop
1972        // for better dispatch on A57.
1973        "subs %w[depth], %w[depth], #16\n"
1974
1975        "smlal2   v12.8h,  v0.16b,  v5.16b\n"
1976        "smlal2   v13.8h,  v1.16b,  v5.16b\n"
1977        "smlal2   v14.8h,  v2.16b,  v5.16b\n"
1978        "smlal2   v15.8h,  v3.16b,  v5.16b\n"
1979
1980        "bne loop_%=\n"
1981
1982        // Final code for the last 16 levels of depth.
1983        // There is nothing to load anymore, only some arithmetic to finish.
1984        "after_loop_last16_%=:\n"
1985
1986        // Some multiplications and 16-bit accumulation were already done above,
1987        // so we start right away in the middle.
1988        "sadalp  v16.4s, v8.8h\n"
1989        "smull    v8.8h,  v0.8b,  v6.8b\n"
1990        "sadalp  v17.4s, v9.8h\n"
1991        "smull    v9.8h,  v1.8b,  v6.8b\n"
1992        "sadalp  v18.4s, v10.8h\n"
1993        "smull    v10.8h,  v2.8b,  v6.8b\n"
1994        "sadalp  v19.4s, v11.8h\n"
1995        "smull    v11.8h,  v3.8b,  v6.8b\n"
1996        "sadalp  v20.4s, v12.8h\n"
1997        "smull    v12.8h,  v0.8b,  v7.8b\n"
1998        "sadalp  v21.4s, v13.8h\n"
```

```
1999        "smull    v13.8h,  v1.8b,  v7.8b\n"
2000        "sadalp  v22.4s, v14.8h\n"
2001        "smull    v14.8h,  v2.8b,  v7.8b\n"
2002        "sadalp  v23.4s, v15.8h\n"
2003        "smull    v15.8h,  v3.8b,  v7.8b\n"
2004
2005        // Multiply-accumulate second-half, again into the same
2006        // 16bit local accumulator registers. This is where we
2007        // take advantage of having int8 instead of uint8 and therefore
2008        // being able to accumulate two products into int16.
2009        "smlal2   v8.8h,  v0.16b,  v6.16b\n"
2010        "smlal2   v9.8h,  v1.16b,  v6.16b\n"
2011        "smlal2   v10.8h,  v2.16b,  v6.16b\n"
2012        "smlal2   v11.8h,  v3.16b,  v6.16b\n"
2013        "smlal2   v12.8h,  v0.16b,  v7.16b\n"
2014        "smlal2   v13.8h,  v1.16b,  v7.16b\n"
2015        "smlal2   v14.8h,  v2.16b,  v7.16b\n"
2016        "smlal2   v15.8h,  v3.16b,  v7.16b\n"
2017
2018        "sadalp  v24.4s, v8.8h\n"
2019        "sadalp  v25.4s, v9.8h\n"
2020        "sadalp  v26.4s, v10.8h\n"
2021        "sadalp  v27.4s, v11.8h\n"
2022        "sadalp  v28.4s, v12.8h\n"
2023        "sadalp  v29.4s, v13.8h\n"
2024        "sadalp  v30.4s, v14.8h\n"
2025        "sadalp  v31.4s, v15.8h\n"
2026
2027        // Reduce 32bit accumulators horizontally, and load
2028        // destination values from memory.
2029        "mov x0, %[accum_ptr]\n"
2030        "addp v0.4s, v16.4s, v20.4s\n"
2031        "addp v1.4s, v24.4s, v28.4s\n"
2032        "ld1 {v12.16b}, [x0], #16\n"
2033        "addp v2.4s, v17.4s, v21.4s\n"
2034        "addp v3.4s, v25.4s, v29.4s\n"
```

```
2035        "ld1 {v13.16b}, [x0], #16\n"
2036        "addp v4.4s, v18.4s, v22.4s\n"
2037        "addp v5.4s, v26.4s, v30.4s\n"
2038        "ld1 {v14.16b}, [x0], #16\n"
2039        "addp v6.4s, v19.4s, v23.4s\n"
2040        "addp v7.4s, v27.4s, v31.4s\n"
2041        "ld1 {v15.16b}, [x0], #16\n"
2042        "mov x0, %[accum_ptr]\n"
2043
2044        // Reduce 32bit accumulators horizontally, second pass
2045        // (each pass adds pairwise. we need to add 4-wise).
2046        "addp v8.4s, v0.4s, v1.4s\n"
2047        "addp v9.4s, v2.4s, v3.4s\n"
2048        "addp v10.4s, v4.4s, v5.4s\n"
2049        "addp v11.4s, v6.4s, v7.4s\n"
2050
2051        // Add horizontally-reduced accumulators into
2052        // the values loaded from memory
2053        "add v12.4s, v12.4s, v8.4s\n"
2054        "add v13.4s, v13.4s, v9.4s\n"
2055        "add v14.4s, v14.4s, v10.4s\n"
2056        "add v15.4s, v15.4s, v11.4s\n"
2057
2058        // Store back into memory
2059        "st1 {v12.16b}, [x0], #16\n"
2060        "st1 {v13.16b}, [x0], #16\n"
2061        "st1 {v14.16b}, [x0], #16\n"
2062        "st1 {v15.16b}, [x0], #16\n"
2063        :  // outputs
2064        [lhs_ptr] "+r"(lhs_ptr), [rhs_ptr] "+r"(rhs_ptr),
2065        [depth] "+r"(depth)
2066        :  // inputs
2067        [accum_ptr] "r"(accum_ptr)
2068        :  // clobbers
2069        "cc", "memory", "x0", "v0", "v1", "v2", "v3", "v4", "v5", "v6",
2070        "v7", "v8", "v9", "v10", "v11", "v12", "v13", "v14", "v15", "v16",
```

```
2071        "v17", "v18", "v19", "v20", "v21", "v22", "v23", "v24", "v25", "v26",
2072        "v27", "v28", "v29", "v30", "v31");
2073    }
2074  };
2075
2076
2077  // We don't actually use int32*int32 in production. This is just an
2078  // experiment to help dissociate the effect of integer-vs-float, from the
2079  // effect of operands width.
2080  struct NEON_64bit_GEMM_Int32_WithScalar {
2081    typedef std::int32_t OperandType;
2082    typedef std::int32_t AccumulatorType;
2083    typedef KernelFormat<KernelSideFormat<CellFormat<4, 1, CellOrder::DepthMajor>, 3>,
2084                         KernelSideFormat<CellFormat<4, 1, CellOrder::DepthMajor>, 2> >
2085        Format;
2086    static void Run(const OperandType* lhs_ptr, const OperandType* rhs_ptr, AccumulatorType* accum_ptr, int depth) {
2087      asm volatile(
2088        // Load accumulators
2089        "mov x0, %[accum_ptr]\n"
2090        "ld1 {v8.16b}, [x0], #16\n"
2091        "ld1 {v16.16b}, [x0], #16\n"
2092        "ld1 {v24.16b}, [x0], #16\n"
2093        "ld1 {v9.16b}, [x0], #16\n"
2094        "ld1 {v17.16b}, [x0], #16\n"
2095        "ld1 {v25.16b}, [x0], #16\n"
2096        "ld1 {v10.16b}, [x0], #16\n"
2097        "ld1 {v18.16b}, [x0], #16\n"
2098        "ld1 {v26.16b}, [x0], #16\n"
2099        "ld1 {v11.16b}, [x0], #16\n"
2100        "ld1 {v19.16b}, [x0], #16\n"
2101        "ld1 {v27.16b}, [x0], #16\n"
2102        "ld1 {v12.16b}, [x0], #16\n"
2103        "ld1 {v20.16b}, [x0], #16\n"
2104        "ld1 {v28.16b}, [x0], #16\n"
2105        "ld1 {v13.16b}, [x0], #16\n"
2106        "ld1 {v21.16b}, [x0], #16\n"
```

```
2107        "ld1 {v29.16b}, [x0], #16\n"
2108        "ld1 {v14.16b}, [x0], #16\n"
2109        "ld1 {v22.16b}, [x0], #16\n"
2110        "ld1 {v30.16b}, [x0], #16\n"
2111        "ld1 {v15.16b}, [x0], #16\n"
2112        "ld1 {v23.16b}, [x0], #16\n"
2113        "ld1 {v31.16b}, [x0], #16\n"
2114
2115        "loop_%=:\n"
2116
2117        // Load 2 Rhs cell of size 1x4 each
2118        "ld1 {v0.4s}, [%[rhs_ptr]], #16\n"
2119        "ld1 {v1.4s}, [%[rhs_ptr]], #16\n"
2120
2121        // Load 3 Lhs cells of size 4x1 each
2122        "ld1 {v2.4s}, [%[lhs_ptr]], #16\n"
2123        "ld1 {v3.4s}, [%[lhs_ptr]], #16\n"
2124        "ld1 {v4.4s}, [%[lhs_ptr]], #16\n"
2125
2126        // Multiply-accumulate
2127        "mla v8.4s, v2.4s, v0.s[0]\n"
2128        "mla v9.4s, v2.4s, v0.s[1]\n"
2129        "mla v10.4s, v2.4s, v0.s[2]\n"
2130        "mla v11.4s, v2.4s, v0.s[3]\n"
2131        "mla v12.4s, v2.4s, v1.s[0]\n"
2132        "mla v13.4s, v2.4s, v1.s[1]\n"
2133        "mla v14.4s, v2.4s, v1.s[2]\n"
2134        "mla v15.4s, v2.4s, v1.s[3]\n"
2135        "mla v16.4s, v3.4s, v0.s[0]\n"
2136        "mla v17.4s, v3.4s, v0.s[1]\n"
2137        "mla v18.4s, v3.4s, v0.s[2]\n"
2138        "mla v19.4s, v3.4s, v0.s[3]\n"
2139        "mla v20.4s, v3.4s, v1.s[0]\n"
2140        "mla v21.4s, v3.4s, v1.s[1]\n"
2141        "mla v22.4s, v3.4s, v1.s[2]\n"
2142        "mla v23.4s, v3.4s, v1.s[3]\n"
```

```
2143        "mla v24.4s, v4.4s, v0.s[0]\n"
2144        "mla v25.4s, v4.4s, v0.s[1]\n"
2145        "mla v26.4s, v4.4s, v0.s[2]\n"
2146        "mla v27.4s, v4.4s, v0.s[3]\n"
2147        "mla v28.4s, v4.4s, v1.s[0]\n"
2148        "mla v29.4s, v4.4s, v1.s[1]\n"
2149        "mla v30.4s, v4.4s, v1.s[2]\n"
2150        "mla v31.4s, v4.4s, v1.s[3]\n"
2151
2152        // Loop. Decrement loop index (depth) by 1, since we just handled 1
2153        // level of depth.
2154        "subs %w[depth], %w[depth], #1\n"
2155        "bne loop_%=\n"
2156
2157        // Store accumulators
2158        "mov x0, %[accum_ptr]\n"
2159        "st1 {v8.16b}, [x0], #16\n"
2160        "st1 {v16.16b}, [x0], #16\n"
2161        "st1 {v24.16b}, [x0], #16\n"
2162        "st1 {v9.16b}, [x0], #16\n"
2163        "st1 {v17.16b}, [x0], #16\n"
2164        "st1 {v25.16b}, [x0], #16\n"
2165        "st1 {v10.16b}, [x0], #16\n"
2166        "st1 {v18.16b}, [x0], #16\n"
2167        "st1 {v26.16b}, [x0], #16\n"
2168        "st1 {v11.16b}, [x0], #16\n"
2169        "st1 {v19.16b}, [x0], #16\n"
2170        "st1 {v27.16b}, [x0], #16\n"
2171        "st1 {v12.16b}, [x0], #16\n"
2172        "st1 {v20.16b}, [x0], #16\n"
2173        "st1 {v28.16b}, [x0], #16\n"
2174        "st1 {v13.16b}, [x0], #16\n"
2175        "st1 {v21.16b}, [x0], #16\n"
2176        "st1 {v29.16b}, [x0], #16\n"
2177        "st1 {v14.16b}, [x0], #16\n"
2178        "st1 {v22.16b}, [x0], #16\n"
```

```
2179        "st1 {v30.16b}, [x0], #16\n"
2180        "st1 {v15.16b}, [x0], #16\n"
2181        "st1 {v23.16b}, [x0], #16\n"
2182        "st1 {v31.16b}, [x0], #16\n"
2183        :  // outputs
2184        [lhs_ptr] "+r"(lhs_ptr), [rhs_ptr] "+r"(rhs_ptr),
2185        [depth] "+r"(depth)
2186        :  // inputs
2187        [accum_ptr] "r"(accum_ptr)
2188        :  // clobbers
2189        "cc", "memory", "x0", "v0", "v1", "v2", "v3", "v4", "v5", "v6",
2190        "v7", "v8", "v9", "v10", "v11", "v12", "v13", "v14", "v15", "v16",
2191        "v17", "v18", "v19", "v20", "v21", "v22", "v23", "v24", "v25", "v26",
2192        "v27", "v28", "v29", "v30", "v31");
2193    }
2194  };
2195
2196  // Not very efficient kernel, just an experiment to see what we can do
2197  // without using NEON multiply-with-scalar instructions.
2198  struct NEON_64bit_GEMM_Float32_WithVectorDuplicatingScalar {
2199    typedef float OperandType;
2200    typedef float AccumulatorType;
2201    typedef KernelFormat<KernelSideFormat<CellFormat<4, 1, CellOrder::DepthMajor>, 3>,
2202                         KernelSideFormat<CellFormat<4, 1, CellOrder::DepthMajor>, 2> >
2203        Format;
2204    static void Run(const OperandType* lhs_ptr, const OperandType* rhs_ptr, AccumulatorType* accum_ptr, int depth) {
2205      asm volatile(
2206        // Load accumulators
2207        "mov x0, %[accum_ptr]\n"
2208        "ld1 {v8.16b}, [x0], #16\n"
2209        "ld1 {v16.16b}, [x0], #16\n"
2210        "ld1 {v24.16b}, [x0], #16\n"
2211        "ld1 {v9.16b}, [x0], #16\n"
2212        "ld1 {v17.16b}, [x0], #16\n"
2213        "ld1 {v25.16b}, [x0], #16\n"
2214        "ld1 {v10.16b}, [x0], #16\n"
```

```
2215        "ld1 {v18.16b}, [x0], #16\n"
2216        "ld1 {v26.16b}, [x0], #16\n"
2217        "ld1 {v11.16b}, [x0], #16\n"
2218        "ld1 {v19.16b}, [x0], #16\n"
2219        "ld1 {v27.16b}, [x0], #16\n"
2220        "ld1 {v12.16b}, [x0], #16\n"
2221        "ld1 {v20.16b}, [x0], #16\n"
2222        "ld1 {v28.16b}, [x0], #16\n"
2223        "ld1 {v13.16b}, [x0], #16\n"
2224        "ld1 {v21.16b}, [x0], #16\n"
2225        "ld1 {v29.16b}, [x0], #16\n"
2226        "ld1 {v14.16b}, [x0], #16\n"
2227        "ld1 {v22.16b}, [x0], #16\n"
2228        "ld1 {v30.16b}, [x0], #16\n"
2229        "ld1 {v15.16b}, [x0], #16\n"
2230        "ld1 {v23.16b}, [x0], #16\n"
2231        "ld1 {v31.16b}, [x0], #16\n"
2232
2233        "loop_%=:\n"
2234
2235        // Load 2 Rhs cell of size 1x4 each
2236        "ld1 {v5.4s}, [%[rhs_ptr]], #16\n"
2237        "ld1 {v6.4s}, [%[rhs_ptr]], #16\n"
2238
2239        // Load 3 Lhs cells of size 4x1 each
2240        "ld1 {v2.4s}, [%[lhs_ptr]], #16\n"
2241        "ld1 {v3.4s}, [%[lhs_ptr]], #16\n"
2242        "ld1 {v4.4s}, [%[lhs_ptr]], #16\n"
2243
2244        // Multiply-accumulate
2245        "dup v0.4s, v5.s[0]\n"
2246        "dup v1.4s, v5.s[1]\n"
2247        "fmla v8.4s, v2.4s, v0.4s\n"
2248        "fmla v16.4s, v3.4s, v0.4s\n"
2249        "fmla v24.4s, v4.4s, v0.4s\n"
2250        "fmla v9.4s, v2.4s, v1.4s\n"
```

```
2251        "fmla v17.4s, v3.4s, v1.4s\n"
2252        "fmla v25.4s, v4.4s, v1.4s\n"
2253        "dup v0.4s, v5.s[2]\n"
2254        "dup v1.4s, v5.s[3]\n"
2255        "fmla v10.4s, v2.4s, v0.4s\n"
2256        "fmla v18.4s, v3.4s, v0.4s\n"
2257        "fmla v26.4s, v4.4s, v0.4s\n"
2258        "fmla v11.4s, v2.4s, v1.4s\n"
2259        "fmla v19.4s, v3.4s, v1.4s\n"
2260        "fmla v27.4s, v4.4s, v1.4s\n"
2261        "dup v0.4s, v6.s[0]\n"
2262        "dup v1.4s, v6.s[1]\n"
2263        "fmla v12.4s, v2.4s, v0.4s\n"
2264        "fmla v20.4s, v3.4s, v0.4s\n"
2265        "fmla v28.4s, v4.4s, v0.4s\n"
2266        "fmla v13.4s, v2.4s, v1.4s\n"
2267        "fmla v21.4s, v3.4s, v1.4s\n"
2268        "fmla v29.4s, v4.4s, v1.4s\n"
2269        "dup v0.4s, v6.s[2]\n"
2270        "dup v1.4s, v6.s[3]\n"
2271        "fmla v14.4s, v2.4s, v0.4s\n"
2272        "fmla v22.4s, v3.4s, v0.4s\n"
2273        "fmla v30.4s, v4.4s, v0.4s\n"
2274        "fmla v15.4s, v2.4s, v1.4s\n"
2275        "fmla v23.4s, v3.4s, v1.4s\n"
2276        "fmla v31.4s, v4.4s, v1.4s\n"
2277
2278        // Loop. Decrement loop index (depth) by 1, since we just handled 1
2279        // level of depth.
2280        "subs %w[depth], %w[depth], #1\n"
2281        "bne loop_%=\n"
2282
2283        // Store accumulators
2284        "mov x0, %[accum_ptr]\n"
2285        "st1 {v8.16b}, [x0], #16\n"
2286        "st1 {v16.16b}, [x0], #16\n"
```

```
2287        "st1 {v24.16b}, [x0], #16\n"
2288        "st1 {v9.16b}, [x0], #16\n"
2289        "st1 {v17.16b}, [x0], #16\n"
2290        "st1 {v25.16b}, [x0], #16\n"
2291        "st1 {v10.16b}, [x0], #16\n"
2292        "st1 {v18.16b}, [x0], #16\n"
2293        "st1 {v26.16b}, [x0], #16\n"
2294        "st1 {v11.16b}, [x0], #16\n"
2295        "st1 {v19.16b}, [x0], #16\n"
2296        "st1 {v27.16b}, [x0], #16\n"
2297        "st1 {v12.16b}, [x0], #16\n"
2298        "st1 {v20.16b}, [x0], #16\n"
2299        "st1 {v28.16b}, [x0], #16\n"
2300        "st1 {v13.16b}, [x0], #16\n"
2301        "st1 {v21.16b}, [x0], #16\n"
2302        "st1 {v29.16b}, [x0], #16\n"
2303        "st1 {v14.16b}, [x0], #16\n"
2304        "st1 {v22.16b}, [x0], #16\n"
2305        "st1 {v30.16b}, [x0], #16\n"
2306        "st1 {v15.16b}, [x0], #16\n"
2307        "st1 {v23.16b}, [x0], #16\n"
2308        "st1 {v31.16b}, [x0], #16\n"
2309        :  // outputs
2310        [lhs_ptr] "+r"(lhs_ptr), [rhs_ptr] "+r"(rhs_ptr),
2311        [depth] "+r"(depth)
2312        :  // inputs
2313        [accum_ptr] "r"(accum_ptr)
2314        :  // clobbers
2315        "cc", "memory", "x0", "v0", "v1", "v2", "v3", "v4", "v5", "v6",
2316        "v7", "v8", "v9", "v10", "v11", "v12", "v13", "v14", "v15", "v16",
2317        "v17", "v18", "v19", "v20", "v21", "v22", "v23", "v24", "v25", "v26",
2318        "v27", "v28", "v29", "v30", "v31");
2319    }
2320  };
2321
2322  // This is the "most natural" kernel, using NEON multiply-with-scalar instructions.
```

```
2323  struct NEON_64bit_GEMM_Float32_WithScalar {
2324    typedef float OperandType;
2325    typedef float AccumulatorType;
2326    typedef KernelFormat<KernelSideFormat<CellFormat<4, 1, CellOrder::DepthMajor>, 3>,
2327                         KernelSideFormat<CellFormat<4, 1, CellOrder::DepthMajor>, 2> >
2328        Format;
2329    static void Run(const OperandType* lhs_ptr, const OperandType* rhs_ptr, AccumulatorType* accum_ptr, int depth) {
2330      asm volatile(
2331        // Load accumulators
2332        "mov x0, %[accum_ptr]\n"
2333        "ld1 {v8.16b}, [x0], #16\n"
2334        "ld1 {v16.16b}, [x0], #16\n"
2335        "ld1 {v24.16b}, [x0], #16\n"
2336        "ld1 {v9.16b}, [x0], #16\n"
2337        "ld1 {v17.16b}, [x0], #16\n"
2338        "ld1 {v25.16b}, [x0], #16\n"
2339        "ld1 {v10.16b}, [x0], #16\n"
2340        "ld1 {v18.16b}, [x0], #16\n"
2341        "ld1 {v26.16b}, [x0], #16\n"
2342        "ld1 {v11.16b}, [x0], #16\n"
2343        "ld1 {v19.16b}, [x0], #16\n"
2344        "ld1 {v27.16b}, [x0], #16\n"
2345        "ld1 {v12.16b}, [x0], #16\n"
2346        "ld1 {v20.16b}, [x0], #16\n"
2347        "ld1 {v28.16b}, [x0], #16\n"
2348        "ld1 {v13.16b}, [x0], #16\n"
2349        "ld1 {v21.16b}, [x0], #16\n"
2350        "ld1 {v29.16b}, [x0], #16\n"
2351        "ld1 {v14.16b}, [x0], #16\n"
2352        "ld1 {v22.16b}, [x0], #16\n"
2353        "ld1 {v30.16b}, [x0], #16\n"
2354        "ld1 {v15.16b}, [x0], #16\n"
2355        "ld1 {v23.16b}, [x0], #16\n"
2356        "ld1 {v31.16b}, [x0], #16\n"
2357
2358        "loop_%=:\n"
```

```
2359
2360        // Load 2 Rhs cell of size 1x4 each
2361        "ld1 {v0.4s}, [%[rhs_ptr]], #16\n"
2362        "ld1 {v1.4s}, [%[rhs_ptr]], #16\n"
2363
2364        // Load 3 Lhs cells of size 4x1 each
2365        "ld1 {v2.4s}, [%[lhs_ptr]], #16\n"
2366        "ld1 {v3.4s}, [%[lhs_ptr]], #16\n"
2367        "ld1 {v4.4s}, [%[lhs_ptr]], #16\n"
2368
2369        // Multiply-accumulate
2370        "fmla v8.4s, v2.4s, v0.s[0]\n"
2371        "fmla v9.4s, v2.4s, v0.s[1]\n"
2372        "fmla v10.4s, v2.4s, v0.s[2]\n"
2373        "fmla v11.4s, v2.4s, v0.s[3]\n"
2374        "fmla v12.4s, v2.4s, v1.s[0]\n"
2375        "fmla v13.4s, v2.4s, v1.s[1]\n"
2376        "fmla v14.4s, v2.4s, v1.s[2]\n"
2377        "fmla v15.4s, v2.4s, v1.s[3]\n"
2378        "fmla v16.4s, v3.4s, v0.s[0]\n"
2379        "fmla v17.4s, v3.4s, v0.s[1]\n"
2380        "fmla v18.4s, v3.4s, v0.s[2]\n"
2381        "fmla v19.4s, v3.4s, v0.s[3]\n"
2382        "fmla v20.4s, v3.4s, v1.s[0]\n"
2383        "fmla v21.4s, v3.4s, v1.s[1]\n"
2384        "fmla v22.4s, v3.4s, v1.s[2]\n"
2385        "fmla v23.4s, v3.4s, v1.s[3]\n"
2386        "fmla v24.4s, v4.4s, v0.s[0]\n"
2387        "fmla v25.4s, v4.4s, v0.s[1]\n"
2388        "fmla v26.4s, v4.4s, v0.s[2]\n"
2389        "fmla v27.4s, v4.4s, v0.s[3]\n"
2390        "fmla v28.4s, v4.4s, v1.s[0]\n"
2391        "fmla v29.4s, v4.4s, v1.s[1]\n"
2392        "fmla v30.4s, v4.4s, v1.s[2]\n"
2393        "fmla v31.4s, v4.4s, v1.s[3]\n"
2394
```

```
2395        // Loop. Decrement loop index (depth) by 1, since we just handled 1
2396        // level of depth.
2397        "subs %w[depth], %w[depth], #1\n"
2398        "bne loop_%=\n"
2399
2400        // Store accumulators
2401        "mov x0, %[accum_ptr]\n"
2402        "st1 {v8.16b}, [x0], #16\n"
2403        "st1 {v16.16b}, [x0], #16\n"
2404        "st1 {v24.16b}, [x0], #16\n"
2405        "st1 {v9.16b}, [x0], #16\n"
2406        "st1 {v17.16b}, [x0], #16\n"
2407        "st1 {v25.16b}, [x0], #16\n"
2408        "st1 {v10.16b}, [x0], #16\n"
2409        "st1 {v18.16b}, [x0], #16\n"
2410        "st1 {v26.16b}, [x0], #16\n"
2411        "st1 {v11.16b}, [x0], #16\n"
2412        "st1 {v19.16b}, [x0], #16\n"
2413        "st1 {v27.16b}, [x0], #16\n"
2414        "st1 {v12.16b}, [x0], #16\n"
2415        "st1 {v20.16b}, [x0], #16\n"
2416        "st1 {v28.16b}, [x0], #16\n"
2417        "st1 {v13.16b}, [x0], #16\n"
2418        "st1 {v21.16b}, [x0], #16\n"
2419        "st1 {v29.16b}, [x0], #16\n"
2420        "st1 {v14.16b}, [x0], #16\n"
2421        "st1 {v22.16b}, [x0], #16\n"
2422        "st1 {v30.16b}, [x0], #16\n"
2423        "st1 {v15.16b}, [x0], #16\n"
2424        "st1 {v23.16b}, [x0], #16\n"
2425        "st1 {v31.16b}, [x0], #16\n"
2426        :  // outputs
2427        [lhs_ptr] "+r"(lhs_ptr), [rhs_ptr] "+r"(rhs_ptr),
2428        [depth] "+r"(depth)
2429        :  // inputs
2430        [accum_ptr] "r"(accum_ptr)
```

```
2431        :  // clobbers
2432        "cc", "memory", "x0", "v0", "v1", "v2", "v3", "v4", "v5", "v6",
2433        "v7", "v8", "v9", "v10", "v11", "v12", "v13", "v14", "v15", "v16",
2434        "v17", "v18", "v19", "v20", "v21", "v22", "v23", "v24", "v25", "v26",
2435        "v27", "v28", "v29", "v30", "v31");
2436    }
2437  };
2438
2439  // Faster kernel contributed by ARM. Tuned for A57.
2440  struct NEON_64bit_GEMM_Float32_WithScalar_A57 {
2441    typedef float OperandType;
2442    typedef float AccumulatorType;
2443    typedef KernelFormat<KernelSideFormat<CellFormat<4, 1, CellOrder::DepthMajor>, 3>,
2444                         KernelSideFormat<CellFormat<4, 1, CellOrder::DepthMajor>, 2> >
2445        Format;
2446    static void Run(const OperandType* lhs_ptr, const OperandType* rhs_ptr, AccumulatorType* accum_ptr, int depth) {
2447      asm volatile(
2448        // Load accumulators
2449        "mov x0, %[accum_ptr]\n"
2450        "ld1 {v8.16b}, [x0], #16\n"
2451        "ld1 {v16.16b}, [x0], #16\n"
2452        "ld1 {v24.16b}, [x0], #16\n"
2453        "ld1 {v9.16b}, [x0], #16\n"
2454        "ld1 {v17.16b}, [x0], #16\n"
2455        "ld1 {v25.16b}, [x0], #16\n"
2456        "ld1 {v10.16b}, [x0], #16\n"
2457        "ld1 {v18.16b}, [x0], #16\n"
2458        "ld1 {v26.16b}, [x0], #16\n"
2459        "ld1 {v11.16b}, [x0], #16\n"
2460        "ld1 {v19.16b}, [x0], #16\n"
2461        "ld1 {v27.16b}, [x0], #16\n"
2462        "ld1 {v12.16b}, [x0], #16\n"
2463        "ld1 {v20.16b}, [x0], #16\n"
2464        "ld1 {v28.16b}, [x0], #16\n"
2465        "ld1 {v13.16b}, [x0], #16\n"
2466        "ld1 {v21.16b}, [x0], #16\n"
```

```
2467        "ld1 {v29.16b}, [x0], #16\n"
2468        "ld1 {v14.16b}, [x0], #16\n"
2469        "ld1 {v22.16b}, [x0], #16\n"
2470        "ld1 {v30.16b}, [x0], #16\n"
2471        "ld1 {v15.16b}, [x0], #16\n"
2472        "ld1 {v23.16b}, [x0], #16\n"
2473        "ld1 {v31.16b}, [x0], #16\n"
2474
2475        // The start of the loop assumes first Rhs cell is already loaded, so
2476        // do it here for first iteration.
2477        "ld1 {v0.4s}, [%[rhs_ptr]], #16\n"
2478
2479        // And the same for the first Lhs cell.
2480        "ld1 {v2.4s}, [%[lhs_ptr]], #16\n"
2481
2482
2483        "loop_%=:\n" // Loop head
2484
2485        // Start the MACs at the head of the loop - 1st cell from each side already loaded.
2486        "fmla v8.4s, v2.4s, v0.s[0]\n"
2487        "fmla v9.4s, v2.4s, v0.s[1]\n"
2488        "ld1 {v1.4s}, [%[rhs_ptr]], #16\n" // Load second Rhs cell.
2489        "fmla v10.4s, v2.4s, v0.s[2]\n"
2490        "fmla v11.4s, v2.4s, v0.s[3]\n"
2491        "ld1 {v3.4s}, [%[lhs_ptr]], #16\n" // Load second Lhs cell.
2492        "fmla v12.4s, v2.4s, v1.s[0]\n"
2493        "fmla v13.4s, v2.4s, v1.s[1]\n"
2494        "ld1 {v4.4s}, [%[lhs_ptr]], #16\n" // Load third Lhs cell.
2495        "fmla v14.4s, v2.4s, v1.s[2]\n"
2496        "fmla v15.4s, v2.4s, v1.s[3]\n"
2497        "ld1 {v2.4s}, [%[lhs_ptr]], #16\n" // Done with first Lhs cell - load for the next iteration early.
2498        "fmla v16.4s, v3.4s, v0.s[0]\n"
2499        "fmla v17.4s, v3.4s, v0.s[1]\n"
2500        "fmla v18.4s, v3.4s, v0.s[2]\n"
2501        "fmla v19.4s, v3.4s, v0.s[3]\n"
2502        "fmla v20.4s, v3.4s, v1.s[0]\n"
```

```
2503        "fmla v21.4s, v3.4s, v1.s[1]\n"
2504        "fmla v22.4s, v3.4s, v1.s[2]\n"
2505        "fmla v23.4s, v3.4s, v1.s[3]\n"
2506        "fmla v24.4s, v4.4s, v0.s[0]\n"
2507        "fmla v25.4s, v4.4s, v0.s[1]\n"
2508        "fmla v26.4s, v4.4s, v0.s[2]\n"
2509        "fmla v27.4s, v4.4s, v0.s[3]\n"
2510        "ld1 {v0.4s}, [%[rhs_ptr]], #16\n" // Done with the first Rhs cell - load for the next iteration early.
2511        "fmla v28.4s, v4.4s, v1.s[0]\n"
2512        "fmla v29.4s, v4.4s, v1.s[1]\n"
2513        // Loop. Decrement loop index (depth) by 1, since we just handled
2514        // 1 level of depth.  Do this a bit before the end of the loop for
2515        // better dispatch on A57.
2516        "subs %w[depth], %w[depth], #1\n"
2517        "fmla v30.4s, v4.4s, v1.s[2]\n"
2518        "fmla v31.4s, v4.4s, v1.s[3]\n"
2519
2520        "bne loop_%=\n"
2521
2522        // Store accumulators
2523        "mov x0, %[accum_ptr]\n"
2524        "st1 {v8.16b}, [x0], #16\n"
2525        "st1 {v16.16b}, [x0], #16\n"
2526        "st1 {v24.16b}, [x0], #16\n"
2527        "st1 {v9.16b}, [x0], #16\n"
2528        "st1 {v17.16b}, [x0], #16\n"
2529        "st1 {v25.16b}, [x0], #16\n"
2530        "st1 {v10.16b}, [x0], #16\n"
2531        "st1 {v18.16b}, [x0], #16\n"
2532        "st1 {v26.16b}, [x0], #16\n"
2533        "st1 {v11.16b}, [x0], #16\n"
2534        "st1 {v19.16b}, [x0], #16\n"
2535        "st1 {v27.16b}, [x0], #16\n"
2536        "st1 {v12.16b}, [x0], #16\n"
2537        "st1 {v20.16b}, [x0], #16\n"
2538        "st1 {v28.16b}, [x0], #16\n"
```

```
2539        "st1 {v13.16b}, [x0], #16\n"
2540        "st1 {v21.16b}, [x0], #16\n"
2541        "st1 {v29.16b}, [x0], #16\n"
2542        "st1 {v14.16b}, [x0], #16\n"
2543        "st1 {v22.16b}, [x0], #16\n"
2544        "st1 {v30.16b}, [x0], #16\n"
2545        "st1 {v15.16b}, [x0], #16\n"
2546        "st1 {v23.16b}, [x0], #16\n"
2547        "st1 {v31.16b}, [x0], #16\n"
2548        :  // outputs
2549        [lhs_ptr] "+r"(lhs_ptr), [rhs_ptr] "+r"(rhs_ptr),
2550        [depth] "+r"(depth)
2551        :  // inputs
2552        [accum_ptr] "r"(accum_ptr)
2553        :  // clobbers
2554        "cc", "memory", "x0", "v0", "v1", "v2", "v3", "v4", "v5", "v6",
2555        "v7", "v8", "v9", "v10", "v11", "v12", "v13", "v14", "v15", "v16",
2556        "v17", "v18", "v19", "v20", "v21", "v22", "v23", "v24", "v25", "v26",
2557        "v27", "v28", "v29", "v30", "v31");
2558    }
2559  };
2560
2561  // Faster kernel contributed by ARM. Tuned for A53.
2562  struct NEON_64bit_GEMM_Float32_WithScalar_A53 {
2563    typedef float OperandType;
2564    typedef float AccumulatorType;
2565    typedef KernelFormat<KernelSideFormat<CellFormat<4, 1, CellOrder::DepthMajor>, 3>,
2566                         KernelSideFormat<CellFormat<4, 1, CellOrder::DepthMajor>, 2> >
2567        Format;
2568    static void Run(const OperandType* lhs_ptr, const OperandType* rhs_ptr, AccumulatorType* accum_ptr, int depth) {
2569      asm volatile(
2570        // Load accumulators
2571        "mov x0, %[accum_ptr]\n"
2572        "ld1 {v8.16b}, [x0], #16\n"
2573        "ld1 {v16.16b}, [x0], #16\n"
2574        "ld1 {v24.16b}, [x0], #16\n"
```

```
2575        "ld1 {v9.16b}, [x0], #16\n"
2576        "ld1 {v17.16b}, [x0], #16\n"
2577        "ld1 {v25.16b}, [x0], #16\n"
2578        "ld1 {v10.16b}, [x0], #16\n"
2579        "ld1 {v18.16b}, [x0], #16\n"
2580        "ld1 {v26.16b}, [x0], #16\n"
2581        "ld1 {v11.16b}, [x0], #16\n"
2582        "ld1 {v19.16b}, [x0], #16\n"
2583        "ld1 {v27.16b}, [x0], #16\n"
2584        "ld1 {v12.16b}, [x0], #16\n"
2585        "ld1 {v20.16b}, [x0], #16\n"
2586        "ld1 {v28.16b}, [x0], #16\n"
2587        "ld1 {v13.16b}, [x0], #16\n"
2588        "ld1 {v21.16b}, [x0], #16\n"
2589        "ld1 {v29.16b}, [x0], #16\n"
2590        "ld1 {v14.16b}, [x0], #16\n"
2591        "ld1 {v22.16b}, [x0], #16\n"
2592        "ld1 {v30.16b}, [x0], #16\n"
2593        "ld1 {v15.16b}, [x0], #16\n"
2594        "ld1 {v23.16b}, [x0], #16\n"
2595        "ld1 {v31.16b}, [x0], #16\n"
2596
2597        // For A53, a very different-looking loop is needed.
2598        //
2599        // The main reason for this is that on A53 128-bit loads take two
2600        // cycles during which no dual issue can occur.  Doing two separate
2601        // 64-bit loads avoids this issue - they each take one cycle and are
2602        // able to dual issue.  Since vector register loads don't dual issue
2603        // with FMLA, we load half the register as normal and the other half
2604        // into an integer register.  This second half can then be moved into
2605        // place later with an INS instruction - which will dual issue with a
2606        // later FP load.
2607        //
2608        // For this kernel there are approximately 3 times as many multiplies
2609        // as loads, so it makes sense to structure the loop into blocks of 4
2610        // cycles, with 1 dedicated "load cycle" and 3 "multiply cycles" per
```

```
2611        // block.  Strictly preserving this structure with NOPs where no load
2612        // is needed seems to result in higher performance.
2613        //
2614        // Choice of x18 to store the upper halves on their way into the
2615        // vector registers is arbitrary.  Added to the clobber list so that
2616        // the compiler will make it available.
2617        //
2618        //
2619        // At the start of the loop, it is assumed that v0 is "half loaded" -
2620        // bottom half in place in d0 and the upper half in x18 ready to
2621        // insert.  So set that up here for the first iteration:
2622        "ldr d0, [%[rhs_ptr]]\n"           // Bottom half of first Rhs cell
2623        "ldr x18, [%[rhs_ptr], #8]\n"      // Upper half
2624        "add %[rhs_ptr], %[rhs_ptr], #16\n"  // Separate increment (needed as there is no operation to load at reg
2625                                                 // + 8 but then increment reg by 16).
2626
2627        // v2 should be fully loaded - as it's outside the loop proper it's fine to use a 128-bit load here.
2628        "ld1 {v2.4s}, [%[lhs_ptr]], #16\n" // first Lhs cell
2629
2630
2631        "loop_%=:\n" // Loop head
2632
2633        // First block of four cycles.  Multplies all require v2 and v0; v2 is
2634        // loaded earlier and v0 is half loaded and completed in the load
2635        // cycle at the start.
2636        "ldr d1, [%[rhs_ptr]]\n"                     // "load" cycle - loading bottom half of v1 (second Rhs cell).
2637        "ins v0.d[1], x18\n"                         // "load" cycle - moving the upper half of v0 into place.
2638        "fmla v8.4s, v2.4s, v0.s[0]\n"               // "fmla" cycle 1 - first multiply.
2639        "ldr x18, [%[rhs_ptr], #8]\n"                // "fmla" cycle 1 - load upper half of v1 into x18.
2640        "fmla v9.4s, v2.4s, v0.s[1]\n"               // "fmla" cycle 2 - second multiply
2641        "add %[rhs_ptr], %[rhs_ptr], #16\n"  // "fmla" cycle 2 - increment Rhs pointer (if needed)
2642        "fmla v10.4s, v2.4s, v0.s[2]\n"              // "fmla" cycle 3 - third multiply.  No more work to dual issue.
2643
2644        // Second block.  Start loading v3 (second Lhs cell), finish loading v1.
2645        "ldr d3, [%[lhs_ptr]]\n"
2646        "ins v1.d[1], x18\n"                             // v1 ready here.
```

```
2647        "fmla v11.4s, v2.4s, v0.s[3]\n"
2648        "ldr x18, [%[lhs_ptr], #8]\n"
2649        "fmla v12.4s, v2.4s, v1.s[0]\n"                    // First use of v1.
2650        "add %[lhs_ptr], %[lhs_ptr], #16\n"
2651        "fmla v13.4s, v2.4s, v1.s[1]\n"
2652
2653        // Third block.  Start loading v4 (third Lhs cell), finish loading v3.
2654        "ldr d4, [%[lhs_ptr]]\n"
2655        "ins v3.d[1], x18\n"                               // v3 ready here.
2656        "fmla v14.4s, v2.4s, v1.s[2]\n"
2657        "ldr x18, [%[lhs_ptr], #8]\n"
2658        "fmla v15.4s, v2.4s, v1.s[3]\n"
2659        "add %[lhs_ptr], %[lhs_ptr], #16\n"
2660        "fmla v16.4s, v3.4s, v0.s[0]\n"                    // First use of v3.
2661
2662        // Fourth block.  v2 (first Lhs cell) is now finished with, so start loading value for next iteration.  Finish loading v4.
2663        "ldr d2, [%[lhs_ptr]]\n"
2664        "ins v4.d[1], x18\n"                               // v4 ready here.
2665        "fmla v17.4s, v3.4s, v0.s[1]\n"
2666        "ldr x18, [%[lhs_ptr], #8]\n"
2667        "fmla v18.4s, v3.4s, v0.s[2]\n"
2668        "add %[lhs_ptr], %[lhs_ptr], #16\n"
2669        "fmla v19.4s, v3.4s, v0.s[3]\n"
2670
2671        // Fifth block, finish loading v2.  No new load to start as the other registers are all still live.
2672        "ins v2.d[1], x18\n"
2673        "fmla v20.4s, v3.4s, v1.s[0]\n"
2674        "fmla v21.4s, v3.4s, v1.s[1]\n"
2675        "fmla v22.4s, v3.4s, v1.s[2]\n"
2676
2677        // Sixth block, nothing to load.  2 nops needed as a single nop would dual issue with the FMLA and break the timing.
2678        "nop\n"
2679        "nop\n"
2680        "fmla v23.4s, v3.4s, v1.s[3]\n"
2681        "fmla v24.4s, v4.4s, v0.s[0]\n"                    // First use of v4.
2682        "fmla v25.4s, v4.4s, v0.s[1]\n"
```

```
2683
2684        // Seventh block, nothing to load.  Decrement the loop counter in this block as the last block is very full.
2685        "nop\n"
2686        "nop\n"
2687        "fmla v26.4s, v4.4s, v0.s[2]\n"
2688        "subs %w[depth], %w[depth], #1\n"
2689        "fmla v27.4s, v4.4s, v0.s[3]\n"
2690        "fmla v28.4s, v4.4s, v1.s[0]\n"
2691
2692        // Eighth block - start loading v0 for next iteration.
2693        "ldr d0, [%[rhs_ptr]]\n"
2694        "fmla v29.4s, v4.4s, v1.s[1]\n"
2695        "ldr x18, [%[rhs_ptr], #8]\n"
2696        "fmla v30.4s, v4.4s, v1.s[2]\n"
2697        "add %[rhs_ptr], %[rhs_ptr], #16\n"
2698        "fmla v31.4s, v4.4s, v1.s[3]\n"
2699
2700        // Loop branch.  This will dual issue in fmla cycle 3 of the 8th block.
2701        "bne loop_%=\n"
2702
2703        // Store accumulators
2704        "mov x0, %[accum_ptr]\n"
2705        "st1 {v8.16b}, [x0], #16\n"
2706        "st1 {v16.16b}, [x0], #16\n"
2707        "st1 {v24.16b}, [x0], #16\n"
2708        "st1 {v9.16b}, [x0], #16\n"
2709        "st1 {v17.16b}, [x0], #16\n"
2710        "st1 {v25.16b}, [x0], #16\n"
2711        "st1 {v10.16b}, [x0], #16\n"
2712        "st1 {v18.16b}, [x0], #16\n"
2713        "st1 {v26.16b}, [x0], #16\n"
2714        "st1 {v11.16b}, [x0], #16\n"
2715        "st1 {v19.16b}, [x0], #16\n"
2716        "st1 {v27.16b}, [x0], #16\n"
2717        "st1 {v12.16b}, [x0], #16\n"
2718        "st1 {v20.16b}, [x0], #16\n"
```

```
2719        "st1 {v28.16b}, [x0], #16\n"
2720        "st1 {v13.16b}, [x0], #16\n"
2721        "st1 {v21.16b}, [x0], #16\n"
2722        "st1 {v29.16b}, [x0], #16\n"
2723        "st1 {v14.16b}, [x0], #16\n"
2724        "st1 {v22.16b}, [x0], #16\n"
2725        "st1 {v30.16b}, [x0], #16\n"
2726        "st1 {v15.16b}, [x0], #16\n"
2727        "st1 {v23.16b}, [x0], #16\n"
2728        "st1 {v31.16b}, [x0], #16\n"
2729        :  // outputs
2730        [lhs_ptr] "+r"(lhs_ptr), [rhs_ptr] "+r"(rhs_ptr),
2731        [depth] "+r"(depth)
2732        :  // inputs
2733        [accum_ptr] "r"(accum_ptr)
2734        :  // clobbers
2735        "cc", "memory", "x0", "x18", "v0", "v1", "v2", "v3", "v4", "v5", "v6",
2736        "v7", "v8", "v9", "v10", "v11", "v12", "v13", "v14", "v15", "v16",
2737        "v17", "v18", "v19", "v20", "v21", "v22", "v23", "v24", "v25", "v26",
2738        "v27", "v28", "v29", "v30", "v31");
2739    }
2740  };
2741
2742  #endif  // __aarch64__
2743
2744  // BEGIN code copied from gemmlowp/internal/kernel_reference.h
2745
2746  // This kernel is templatized in an arbitrary Format template parameter,
2747  // allowing it to have any arbitrary format.
2748  template <typename tOperandType, typename tAccumulatorType, typename tFormat>
2749  struct ReferenceKernel {
2750    typedef tOperandType OperandType;
2751    typedef tAccumulatorType AccumulatorType;
2752    typedef tFormat Format;
2753
2754    static void Run(const OperandType* lhs_ptr, const OperandType* rhs_ptr, AccumulatorType* accum_ptr, int depth) {
```

```
2755      const int depth_cells = static_cast<int>(depth / Format::kDepth);
2756
2757      // The outer loop is over the depth dimension.
2758      for (int dc = 0; dc < depth_cells; dc++) {
2759        // The next two loops are over cells of the Lhs (stacked vertically),
2760        // and over cells of the Rhs (stacked horizontally).
2761        for (int rc = 0; rc < Format::Lhs::kCells; rc++) {
2762          const OperandType* lhs_cell_ptr = lhs_ptr +
2763                                            (dc * Format::Lhs::kCells + rc) *
2764                                                Format::Lhs::Cell::kWidth *
2765                                                Format::kDepth;
2766          for (int cc = 0; cc < Format::Rhs::kCells; cc++) {
2767            const OperandType* rhs_cell_ptr = rhs_ptr +
2768                                              (dc * Format::Rhs::kCells + cc) *
2769                                                  Format::Rhs::Cell::kWidth *
2770                                                  Format::kDepth;
2771
2772            // Now we are inside one cell of the Lhs and inside one cell
2773            // of the Rhs, so the remaining inner loops are just
2774            // traditional three loops of matrix multiplication.
2775            for (int di = 0; di < Format::kDepth; di++) {
2776              for (int ri = 0; ri < Format::Lhs::Cell::kWidth; ri++) {
2777                for (int ci = 0; ci < Format::Rhs::Cell::kWidth; ci++) {
2778                  const OperandType* lhs_coeff_ptr =
2779                      lhs_cell_ptr +
2780                      OffsetIntoCell<typename Format::Lhs::Cell>(ri, di);
2781                  const OperandType* rhs_coeff_ptr =
2782                      rhs_cell_ptr +
2783                      OffsetIntoCell<typename Format::Rhs::Cell>(ci, di);
2784                  AccumulatorType* accumulator_coeff_ptr =
2785                      accum_ptr + (ri + rc * Format::Lhs::Cell::kWidth) +
2786                      (ci + cc * Format::Rhs::Cell::kWidth) * Format::kRows;
2787                  *accumulator_coeff_ptr +=
2788                      AccumulatorType(*lhs_coeff_ptr) * AccumulatorType(*rhs_coeff_ptr);
2789                }
2790              }
```

```
2791            }
2792          }
2793        }
2794      }
2795    }
2796  };
2797
2798  // END code copied from gemmlowp/internal/kernel_reference.h
2799
2800  template <typename DataType>
2801  class CacheLineAlignedBuffer
2802  {
2803  public:
2804    CacheLineAlignedBuffer(std::size_t size)
2805      : size_(size) {
2806      data_ = nullptr;
2807      posix_memalign(reinterpret_cast<void**>(&data_), kCacheLineSize, size_ * sizeof(DataType));
2808    }
2809
2810    ~CacheLineAlignedBuffer() {
2811      free(data_);
2812    }
2813
2814    const DataType *data() const { return data_; }
2815    DataType *data() { return data_; }
2816
2817    const std::size_t size() const { return size_; }
2818
2819  private:
2820    const std::size_t size_;
2821    DataType *data_;
2822  };
2823
2824  template <typename DataType>
2825  void FillRandom(CacheLineAlignedBuffer<DataType>* buffer) {
2826    static std::mt19937 generator(0);
```

```
2827    // 100 is smaller than any nonzero bound of the range of any data type.
2828    const DataType kMaxVal = DataType(100);
2829    const DataType kMinVal = std::is_signed<DataType>::value ? -kMaxVal : DataType(0);
2830    std::uniform_real_distribution<float> dist(kMinVal, kMaxVal);
2831    for (std::size_t i = 0; i < buffer->size(); i++) {
2832      buffer->data()[i] = DataType(dist(generator));
2833    }
2834  }
2835
2836
2837  template <typename DataType>
2838  void FillZero(CacheLineAlignedBuffer<DataType>* buffer) {
2839    for (std::size_t i = 0; i < buffer->size(); i++) {
2840      buffer->data()[i] = DataType(0);
2841    }
2842  }
2843
2844  template <typename DataType>
2845  void Copy(CacheLineAlignedBuffer<DataType>* dst, const CacheLineAlignedBuffer<DataType>& src) {
2846    assert(dst->size() == src.size());
2847    memcpy(dst->data(), src.data(), src.size() * sizeof(DataType));
2848  }
2849
2850  template <typename DataType>
2851  void PrintMatrix(int rows, int cols, int rowstride, int colstride, const DataType* data) {
2852    for (int r = 0; r < rows; r++) {
2853      for (int c = 0; c < cols; c++) {
2854        std::cerr << double(data[r * rowstride + c * colstride]) << " ";
2855      }
2856      std::cerr << std::endl;
2857    }
2858    std::cerr << std::endl;
2859  }
2860
2861  template <typename DataType>
2862  bool approx_equals(DataType a, DataType b) {
```

```
2863    return a == b;
2864  }
2865
2866  template <>
2867  bool approx_equals(float a, float b) {
2868    if (!a && !b) {
2869      return true;
2870    }
2871    return std::abs(a - b) < 1e-3f * std::min(std::abs(a), std::abs(b));
2872  }
2873
2874  template <typename Kernel>
2875  void test_kernel(int depth, const char* kernel_name)
2876  {
2877    typedef typename Kernel::OperandType OperandType;
2878    typedef typename Kernel::AccumulatorType AccumulatorType;
2879    typedef typename Kernel::Format Format;
2880    static const int kLhsWidth = Format::Lhs::kWidth;
2881    static const int kRhsWidth = Format::Rhs::kWidth;
2882
2883    typedef ReferenceKernel<OperandType, AccumulatorType, Format>
2884      ReferenceKernel;
2885
2886    CacheLineAlignedBuffer<OperandType> lhs(kLhsWidth * depth);
2887    CacheLineAlignedBuffer<OperandType> rhs(kRhsWidth * depth);
2888    CacheLineAlignedBuffer<AccumulatorType> accum_initial(kLhsWidth * kRhsWidth);
2889    CacheLineAlignedBuffer<AccumulatorType> accum(kLhsWidth * kRhsWidth);
2890    CacheLineAlignedBuffer<AccumulatorType> accum_reference(kLhsWidth * kRhsWidth);
2891
2892    FillRandom(&lhs);
2893    FillRandom(&rhs);
2894    FillRandom(&accum_initial);
2895    Copy(&accum, accum_initial);
2896    Copy(&accum_reference, accum_initial);
2897
2898
```

```
2899    ReferenceKernel::Run(lhs.data(), rhs.data(), accum_reference.data(), depth);
2900    Kernel::Run(lhs.data(), rhs.data(), accum.data(), depth);
2901
2902    for (int l = 0; l < kLhsWidth; l++) {
2903      for (int r = 0; r < kRhsWidth; r++) {
2904        const int index = l + kLhsWidth * r;
2905        if (!approx_equals(accum.data()[index], accum_reference.data()[index])) {
2906          std::cerr << "Arithmetic error in kernel:" << std::endl << "    " <<
2907              kernel_name << std::endl <<
2908              "Wrong accumulator for depth=" << depth << ", " <<
2909              "at l = " << l << ", r = " << r << std::endl;
2910          std::cerr << "reference value: " << accum_reference.data()[index] << std::endl;
2911          std::cerr << "actual value:    " << accum.data()[index] << std::endl;
2912          if (depth <= 16) {
2913            std::cerr << "LHS matrix:" << std::endl;
2914            PrintMatrix(kLhsWidth, depth, 1, kLhsWidth, lhs.data());
2915            std::cerr << "RHS matrix:" << std::endl;
2916            PrintMatrix(depth, kRhsWidth, kRhsWidth, 1, rhs.data());
2917            std::cerr << "Initial Accumulator matrix:" << std::endl;
2918            PrintMatrix(kLhsWidth, kRhsWidth, 1, kLhsWidth, accum_initial.data());
2919            std::cerr << "Reference Accumulator matrix:" << std::endl;
2920            PrintMatrix(kLhsWidth, kRhsWidth, 1, kLhsWidth, accum_reference.data());
2921            std::cerr << "Actual Accumulator matrix:" << std::endl;
2922            PrintMatrix(kLhsWidth, kRhsWidth, 1, kLhsWidth, accum.data());
2923          }
2924          abort();
2925        }
2926      }
2927    }
2928  }
2929
2930  template <typename Kernel>
2931  int ops(int depth) {
2932    // 2x the number of multiply-accumulate scalar ops.
2933    return 2 *
2934        Kernel::Format::Lhs::kWidth *
```

```
2935        Kernel::Format::Rhs::kWidth *
2936        depth;
2937  }
2938
2939  template <unsigned Modulus, typename Integer>
2940  Integer RoundDown(Integer i) {
2941    return i - (i % Modulus);
2942  }
2943
2944  int CacheSizeInKB() {
2945    static const char* cache_size_k_env = getenv("CACHE_SIZE_KB");
2946    static const int cache_size_k =
2947        cache_size_k_env ? atoi(cache_size_k_env) : kDefaultCacheSizeK;
2948    return cache_size_k;
2949  }
2950
2951  template <typename Kernel>
2952  int BenchmarkDepthToFitInCache() {
2953    const int cache_size_bytes = 1024 * CacheSizeInKB();
2954
2955    // Subtract the typical size of a few cache lines, so
2956    // we don't need to worry too hard about e.g. some stack data.
2957    const int conservative_cache_size_bytes =
2958        cache_size_bytes - 2 * kCacheLineSize;
2959
2960    // We will subtract the memory occupied by accumulators.
2961    typedef typename Kernel::AccumulatorType AccumulatorType;
2962    const int kAccumulatorBytes =
2963        sizeof(AccumulatorType) * Kernel::Format::Lhs::kWidth * Kernel::Format::Rhs::kWidth;
2964
2965    // Compute the depth.
2966    typedef typename Kernel::OperandType OperandType;
2967    const int kBytesPerUnitOfDepth =
2968        sizeof(OperandType) * (Kernel::Format::Lhs::kWidth + Kernel::Format::Rhs::kWidth);
2969    const int unrounded_depth =
2970        (conservative_cache_size_bytes - kAccumulatorBytes) / kBytesPerUnitOfDepth;
```

```
2971
2972    // Cap depth, to avoid unfairly favoring narrower kernels
2973    const int kMaxDepth = 1024;
2974    const int clamped_unrounded_depth = std::min(kMaxDepth, unrounded_depth);
2975
2976    // Round depth down to a multiple of cache line size, which helps because
2977    // our kernels may crash if depth is not a multiple of the number of
2978    // depth level that they want to
2979    // handle at each loop iteration, and we don't want to require kernels
2980    // to be more complex. Currently all kernels process 1, 2 or 8 levels of
2981    // depth at a time. The main reason why that might increase in the future
2982    // is if registers get wider, but I don't suppose that register could
2983    // ever get wider than cache lines.
2984    return RoundDown<kCacheLineSize>(clamped_unrounded_depth);
2985  }
2986
2987  double current_time_in_seconds() {
2988    timespec t;
2989    clock_gettime(CLOCK_THREAD_CPUTIME_ID, &t);
2990    return t.tv_sec + 1e-9 * t.tv_nsec;
2991  }
2992
2993  template <typename Kernel>
2994  double benchmark() {
2995    // Minimum duration for this benchmark to run. If the workload finishes
2996    // sooner, we retry with double the number of iterations.
2997    static const double min_benchmark_time_in_seconds = 0.5;
2998
2999    const int depth = BenchmarkDepthToFitInCache<Kernel>();
3000
3001    typedef typename Kernel::OperandType OperandType;
3002    typedef typename Kernel::AccumulatorType AccumulatorType;
3003
3004    CacheLineAlignedBuffer<OperandType> lhs(Kernel::Format::Lhs::kWidth * depth);
3005    CacheLineAlignedBuffer<OperandType> rhs(Kernel::Format::Rhs::kWidth * depth);
3006    CacheLineAlignedBuffer<AccumulatorType> accum(Kernel::Format::Lhs::kWidth * Kernel::Format::Rhs::kWidth);
```

```
3007
3008    std::uint64_t iters_at_a_time = 1;
3009
3010    for (std::uint64_t iters_at_a_time = 1; ; iters_at_a_time *= 2) {
3011      const double t_start = current_time_in_seconds();
3012      for (std::uint64_t i = 0; i < iters_at_a_time; i++) {
3013        Kernel::Run(lhs.data(), rhs.data(), accum.data(), depth);
3014      }
3015      const double t_end = current_time_in_seconds();
3016      const double elapsed = t_end - t_start;
3017      if (elapsed > min_benchmark_time_in_seconds) {
3018        return iters_at_a_time * ops<Kernel>(depth) / elapsed;
3019      }
3020    }
3021  }
3022
3023  int get_num_cpus() {
3024    static const int n = sysconf(_SC_NPROCESSORS_CONF);
3025    return n;
3026  }
3027
3028  #ifdef PRINT_CPUFREQ
3029  void maybe_print_one_word_file(const std::string& filename) {
3030    std::ifstream file(filename);
3031    if (file.fail()) {
3032      // fail silently, the Android /sys filesystem might
3033      // not be universal...
3034      return;
3035    }
3036    std::string word;
3037    file >> word;
3038    std::cout << filename << ": " << word << std::endl;
3039  }
3040
3041  void print_current_cpufreq(int cpu) {
3042    std::stringstream dir_stream;
```

```
3043    dir_stream << "/sys/devices/system/cpu/cpu" << cpu << "/cpufreq/";
3044    std::string dir;
3045    dir_stream >> dir;
3046    maybe_print_one_word_file(dir + "cpuinfo_cur_freq");
3047    maybe_print_one_word_file(dir + "scaling_cur_freq");
3048  }
3049  #endif
3050
3051  template <typename Kernel>
3052  void benchmark_and_print_results(const char* kernel_name) {
3053    test_kernel<Kernel>(Kernel::Format::kDepth, kernel_name);
3054    test_kernel<Kernel>(2 * Kernel::Format::kDepth, kernel_name);
3055    test_kernel<Kernel>(1024, kernel_name);
3056    const int num_cpus = get_num_cpus();
3057    for (int cpu = 0; cpu < num_cpus; cpu++) {
3058      cpu_set_t s;
3059      CPU_ZERO(&s);
3060      CPU_SET(cpu, &s);
3061      sched_setaffinity(0, sizeof(cpu_set_t), &s);
3062
3063      std::cout << kernel_name <<
3064          "(depth=" << BenchmarkDepthToFitInCache<Kernel>() <<
3065          ") on CPU #" << cpu << ": " <<
3066          benchmark<Kernel>() * 1e-9f << " Gop/s" << std::endl;
3067
3068  #ifdef PRINT_CPUFREQ
3069      print_current_cpufreq(cpu);
3070  #endif
3071    }
3072  }
3073
3074  #define BENCHMARK(Kernel) \
3075      do { \
3076        benchmark_and_print_results<Kernel>(#Kernel); \
3077      } while (false)
3078
```

```
3079  int main() {
3080    std::cout << "There are " << get_num_cpus() << " CPU cores." << std::endl;
3081    std::cout << "Targeting a cache size of " << CacheSizeInKB() << " K" << std::endl;
3082
3083  #ifdef __arm__
3084    std::cout << "CPU architecture: ARM 32bit" << std::endl;
3085    BENCHMARK(NEON_32bit_GEMM_Uint8Operands_Uint32Accumulators);
3086    BENCHMARK(NEON_32bit_GEMM_Uint8Operands_Uint32Accumulators_noexpand);
3087    BENCHMARK(NEON_32bit_GEMM_Int32_WithScalar);
3088    BENCHMARK(NEON_32bit_GEMM_Float32_MLA_WithVectorDuplicatingScalar);
3089  #ifdef __ARM_FEATURE_FMA
3090    BENCHMARK(NEON_32bit_GEMM_Float32_FMA_WithVectorDuplicatingScalar);
3091  #endif
3092    BENCHMARK(NEON_32bit_GEMM_Float32_MLA_WithScalar);
3093    BENCHMARK(NEON_32bit_GEMM_Float32_WithScalar_A53);
3094    BENCHMARK(NEON_32bit_GEMM_Float32_WithScalar_A53_depth2);
3095    BENCHMARK(NEON_32bit_GEMM_Float32_MLA_Rotating);
3096  #ifdef __ARM_FEATURE_FMA
3097    BENCHMARK(NEON_32bit_GEMM_Float32_FMA_Rotating);
3098  #endif
3099  #endif
3100
3101  #ifdef __aarch64__
3102    std::cout << "CPU architecture: ARM 64bit" << std::endl;
3103    BENCHMARK(NEON_64bit_GEMM_Int8Operands_Int32Accumulators_AccumTwoWithin16Bits);
3104    BENCHMARK(NEON_64bit_GEMM_Uint8Operands_Uint32Accumulators);
3105    BENCHMARK(NEON_64bit_GEMM_Uint8Operands_Uint32Accumulators_noexpand_A57);
3106    BENCHMARK(NEON_64bit_GEMM_Int32_WithScalar);
3107    BENCHMARK(NEON_64bit_GEMM_Float32_WithVectorDuplicatingScalar);
3108    BENCHMARK(NEON_64bit_GEMM_Float32_WithScalar);
3109    BENCHMARK(NEON_64bit_GEMM_Float32_WithScalar_A57);
3110    BENCHMARK(NEON_64bit_GEMM_Float32_WithScalar_A53);
3111  #endif
3112  }
```