The Journal of Supercomputing (2021) 77:11257–11269
https://doi.org/10.1007/s11227-021-03636-4

Low precision matrix multiplication for efficient deep learning in NVIDIA Carmel processors

Pablo San Juan[2] · Rafael Rodríguez-Sánchez[1] · Francisco D. Igual[1] · Pedro Alonso-Jordá[2] · Enrique S. Quintana-Ortí[2]

Accepted: 13 January 2021 / Published online: 22 March 2021

Abstract

We introduce a high performance, multi-threaded realization of the GEMM kernel for the ARMv8.2 architecture that operates with 16-bit (half precision)/queryKindly check and confirm whether the corresponding author is correctly identified. floating point operands. Our code is especially designed for efficient machine learning inference (and to a certain extent, also training) with deep neural networks. The results on the NVIDIA Carmel multicore processor, which implements the ARMv8.2 architecture, show considerable performance gains for the GEMM kernel, close to the theoretical peak acceleration that could be expected when moving from 32-bit arithmetic/data to 16-bit. Combined with the type of convolution operator arising in convolutional neural networks, the speed-ups are more modest though still relevant.

Keywords Deep learning · Matrix multiplication · High performance · NVIDIA Carmel system-on-chip (SoC)

✉ Pablo San Juan
p.sanjuan@upv.es

Rafael Rodríguez-Sánchez
rafaelrs@ucm.es

Francisco D. Igual
figual@ucm.es

Pedro Alonso-Jordá
palonso@upv.es

Enrique S. Quintana-Ortí
quintana@disca.upv.es

1 Universidad Complutense de Madrid, Madrid, Spain

2 Universitat Politècnica de València, Valencia, Spain

1 Introduction

Machine learning via deep neural networks (that is, deep learning) is being increasingly adopted beyond traditional computer vision niches to gain momentum, in the last years, in natural language processing (NLP), recommendation systems, and a variety of scientific and engineering applications [1–4].

Deep neural networks (DNNs) consist of an ensemble of “layers”, each composed of multiple “neurons”. Each layer receives an input, processes it via a nonlinear transform, and passes the result to the next layer. Convolutional neural networks (CNNs) include specialized layers that pay attention to neighbour inputs via convolution filters. More sophisticated designs, such as recurrent neural networks (RNNs), residual neural networks, transformers, etc. include “connections” between non-consecutive layers [5,6].

Supervised deep learning commences with the training of the DNN. To this end, the network is (repeatedly) exposed to a (large) collection of input-output pairs that are used to “learn” by adjusting its parameters (weights). This is achieved via an optimization algorithm such as the Stochastic Gradient Descent (SGD) or some related variant [5]. This first stage is followed by the inference, where the DNN is put into production mode, working on “unseen” data. Self-supervised transformers for NLP present a pre-training+fine-tuning stage prior to inference [4,6].

There are significant differences between the training and inference stages: the former involves complex mathematics and requires floating point arithmetic. As a result, training is computationally expensive and is performed off-line, usually on high performance facilities equipped with massively parallel processors (typically graphics processing units, GPUs). In contrast, inference involves much cheaper computations, and in general requires low precision floating point or even integer arithmetic. However, this second stage often presents strict time constraints and is conducted in low-power systems-on-chip (SoCs) with relatively low computational capacity.

The general matrix multiplication (GEMM) is a fundamental kernel, both for training and inference, in most types of DNNs. For CNNs, for example, the convolutional operators, which are the computational keystone for this type of networks, can be cast in terms of large GEMM kernels via the IM2COL transform [7]. For performance reasons, when training/inference is applied simultaneously to groups of several input vectors (batches), multilayer perceptrons (MLPs), transformers and other types of DNNs also require the execution of large GEMM kernels.

In this paper, we contribute to the efficient execution of deep learning algorithms on the ARM-based NVIDIA Carmel multicore processor present in the NVIDIA Jetson AGX Xavier SoC. For this purpose, we depart from the BLIS infrastructure [8] to design and implement a new high-performance implementation of the GEMM kernel, tuned for the 8-core NVIDIA Carmel (ARMv8.2) processor, which operates with data in 16-bit floating point arithmetic. This datatype format offers a good compromise for the precision required in training (32- or 16-bit floating point) and inference (16- or 8-bit floating point or integer arithmetic).

The rest of the paper is structured as follows. In Sect. 2 we review the BLIS approach to high performance GEMM, and in Sect. 3 we present our architecture-specific optimization of this kernel. In Sect. 4 we offer a complete evaluation of GEMM on the

```
L1  for $j_c = 0, 1, \ldots, n-1$ in steps of $n_c$ do
L2    for $p_c = 0, 1, \ldots, k-1$ in steps of $k_c$ do
        $B(p_c : p_c + k_c - 1, j_c : j_c + n_c - 1) \rightarrow B_c$          // Packing into $B_c$
L3      for $i_c = 0, 1, \ldots, m-1$ in steps of $m_c$ do
          $A(i_c : i_c + m_c - 1, p_c : p_c + k_c - 1) \rightarrow A_c$        // Packing into $A_c$
L4        for $j_r = 0, 1, \ldots, n_c - 1$ in steps of $n_r$ do
L5          for $i_r = 0, 1, \ldots, m_c - 1$ in steps of $m_r$ do
              $C_c(i_r : i_r + m_r - 1, j_r : j_r + n_r - 1)$                // Micro-kernel
                  $\mathrel{+}= A_c(i_r : i_r + m_r - 1, 0 : k_c - 1)$
                  $\cdot \quad B_c(0 : k_c - 1, j_r : j_r + n_r - 1)$
            end for
          end for
        end for
      end for
    end for
```

Fig. 1 High performance BLIS implementation of the GEMM operation: $C \mathrel{+}= A \cdot B$. The dimensions of the operands are $C \rightarrow m \times n$, $A \rightarrow m \times k$, and $B \rightarrow k \times n$. In the code, $C_c \equiv C(i_c : i_c + m_c - 1,\ j_c : j_c + n_c - 1)$ is just a notation artifact, introduced to ease the presentation of the algorithm, while A_c, B_c correspond to actual buffers that are involved in data copies

NVIDIA Carmel processor both as a standalone kernel and in the context of an integrated convolution operator for CNNs. The paper is closed in Sect. 5 with a summary and a few concluding remarks.

2 Overview of BLIS GEMM

The *Basic Linear Algebra Instantiation Software* (BLIS) is an infrastructure developed at The University of Texas at Austin that facilitates the rapid development of *Basic Linear Algebra Suprograms* (BLAS) [9]. As exposed in the previous section, in our work for DNNs we are interested in obtaining a high-performance realization of GEMM that operates in 16-bit floating-point precision on the 8-core NVIDIA Carmel processor. For this purpose, we exploit the high-level formulation of GEMM in BLIS, which offers a generic implementation of this kernel as five nested loops embedding two packing routines and a micro-kernel; see Fig. 1. The micro-kernel is encoded as a loop around a rank–1 update and comprises all the arithmetic operations. The packing routines re-arrange small blocks of data from the input operands A and B into two buffers, A_c and B_c, respectively, to ensure unit stride access to the contents of these buffers during the execution of the micro-kernel.

BLIS follows the design of GotoBLAS [10] to attain high performance on current processor architectures but differs in a substantial aspect. Concretely, most of the BLIS realization is implemented in a high-level, portable language such as C so that, in order to deliver high performance on a particular processor, BLIS only requires 1) a tuned and architecture-specific implementation of the micro-kernel, which in general is vectorized using assembly [8]; and 2) an appropriate selection of the loop strides m_c, n_c, k_c that matches the cache configuration of the target processor; see [8,11] for details. (Compared with other work [12], BLIS sacrifices some portability in exchange for higher performance by relying on an architecture-specific micro-kernel, in general developed in assembly code.)

In a multi-threaded implementation of BLIS, the loops of the GEMM kernel to be parallelized can be selected at execution time. Several studies [13–15] show that, in general, loop L1 is a good candidate for multi-socket platforms with on-chip L3 caches; loop L3 should be parallelized when each core has its own L2 cache; and loops L4 and L5 are convenient choices if the cores share the L2 cache. However, when the matrix operands present one or more small dimensions (that is, values for m, n, and/or k) compared with the number of cores, the specific loops that will offer higher performance when parallelized strongly depend on these dimensions.

3 Optimized GEMM for NVIDIA Carmel processor

The new GEMM mimics the BLIS approach (see Sect. 2) to provide an optimized 16-bit (or half) floating point precision GEMM kernel for the NVIDIA Carmel processor. In this section, we describe our optimized micro-kernel for half precision (HP) using the 16-bit floating point (FP16) instructions in the ARM instruction set architecture (ISA). As baseline for our implementation, we leveraged the single precision (SP) microkernel included in BLIS for the ARM Cortex-A57, because this CPU realizes the same ARMv8 architecture that is present in the NVIDIA Carmel processor. In addition, we illustrate the tuning of the cache configuration parameters (m_c, n_c, k_c) for the NVIDIA Carmel processor since its cache organization differs from that of the ARM Cortex-A57 and, therefore, the parameters selected in BLIS for the latter are not necessarily optimal.

3.1 Reference micro-kernel

Let us consider the small GEMM performed by the micro-kernel:

$$\begin{aligned} C_r \mathrel{+}= A_r \cdot B_r \equiv C_c(i_r : i_r + m_r - 1, j_r : j_r + n_r - 1) \\ \mathrel{+}= A_c(i_r : i_r + m_r - 1, 0 : k_c - 1) \\ \cdot\ B_c(0 : k_c - 1, j_r : j_r + n_r - 1), \end{aligned}$$

which involves the micro-tiles C_r, A_r and B_r, of dimensions $m_r \times n_r$, $m_r \times k_c$ and $k_c \times n_r$, respectively. As argued in Sect. 2, this micro-kernel is realized in practice (in assembly code) as an additional loop which, at each iteration, performs a rank–1 update of the full C_r using one column of A_r and one row of B_r (that is, an outer product) as illustrated next:

```
L6  1: for $k_r = 0, 1, \ldots, k_c - 1$ do
    2:     $C_r \mathrel{+}= A_r(0 : m_r - 1, k_r) \cdot B_r(k_r, 0 : n_r - 1)$
    3: end for
```

BLIS offers a portable realization of GEMM that includes a baseline implementation of the micro-kernel in C. Although properly adjusting the cache configuration parameters m_c, n_c, k_c for this reference implementation can result in a fair utilization of the memory hierarchy of the target processor, this basic instance will usually render low performance because it lacks the elaborate optimizations such as loop unrolling and/or use of vector instructions [16]. Furthermore, BLIS only offers this generic realization of GEMM for the standard 32-bit (SP) and 64-bit (double precision) data types.

3.2 Optimized HP micro-kernel

In order to design an optimized HP micro-kernel we took into account the following relevant capabilities of the target processor architecture (ARMv8):

- The architecture comprises 32 128-bit wide SIMD registers (denoted in the following as $v_0 - v_{31}$).
- Each SIMD register can hold up to 8 HP floating point numbers (FP16).
- Vector loads and vector stores are supported in hardware for FP16.
- Compute capabilities for FP16 are available with the ARMV8.2-FP16 architecture extension.

Armed with this information, we used ARM assembly instructions to develop a micro-kernel with the following characteristics, that aims to maximize register usage without incurring in register spilling:

- The "dimension" of the micro-kernel is $m_r \times n_r = 24 \times 8$.
- 24 vector registers are used to hold partial contributions to C_r.
- 6/2 vector registers used to hold two columns/rows of A_r / B_r.
- The iteration loop of the micro-kernel (over the k_c dimension of the problem) is unrolled with a factor of 4.
- The computation is performed using the FP16 FMLA (floating point fused multiply-add to accumulator) vector instructions available in ARMv8.2. In more detail, we leverage the version which multiplies all entries of v_{src1} by an element of v_{src2}, and adds all the components of the result to those of v_{dst}. This version is referred in the following as *FMLA by element*.
- Loads and stores are performed using multiple structure SIMD instructions `LD1` and `ST1`, respectively. The columns of A_r and the columns of C_r are loaded/stored in blocks of 3 vectors (24 elements). `LD1`/`ST1` allow to load/store up to 4 vector registers with a single instruction, respectively, though we leveraged them to load/store 3 vector registers at a time.
- C_r is assumed to be stored in column major-order.
- Figure 2 shows the register mapping of the contribution to C_r, and the buffers A_r and B_r. This mapping specifies the vector registers that are used to accumulate the updates over C_r ($v_8 - v_{31}$) and the vectors used to load the columns of A_r ($v_0 - v_5$) and B_r ($v_6 - v_7$). Each vector register is depicted in a different color to improve readability. At each iteration of the k_c loop, all registers of the intermediate result C_r are updated once using one column of A_r and one row of B_r. Each column/row of A_r/B_r is loaded in different registers to allow load and computation overlap.

Algorithm 1 illustrates the structure of the micro-kernel prior to loop unrolling and additional fine-grain optimizations. For simplicity, the algorithm targets the specific case of the GEMM kernel $C = A \cdot B$ (but our actual implementation covers the general case $C = \beta C + \alpha A \cdot B$, for any two scalars α, β). The iteration loop updates the

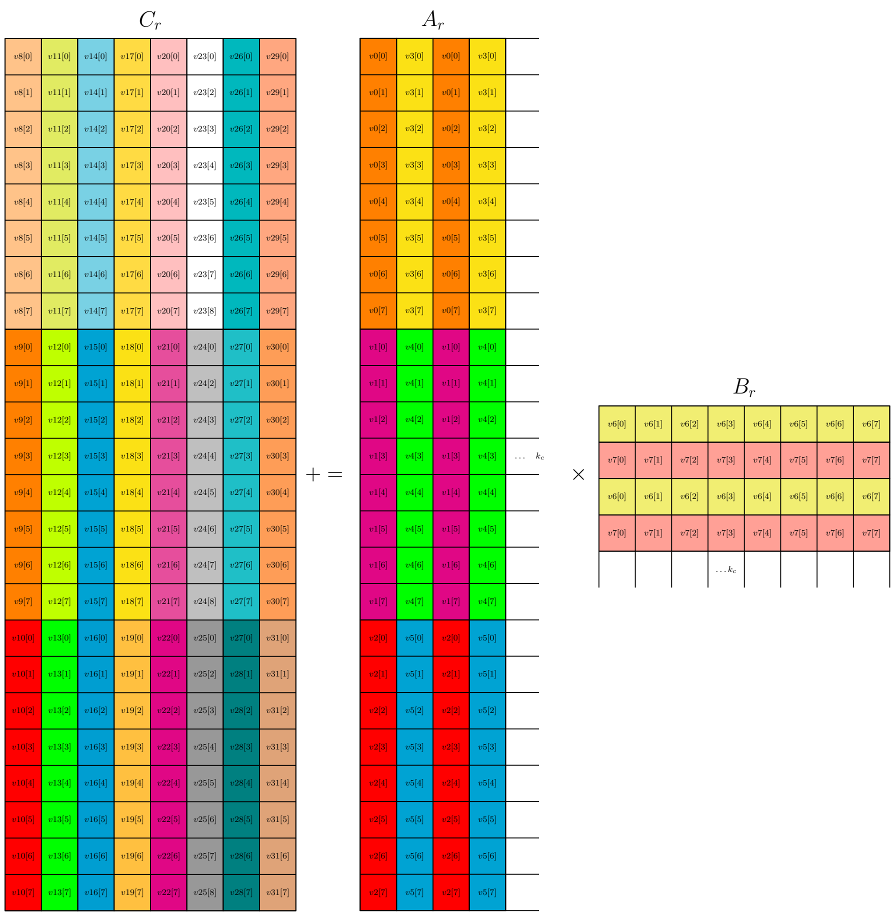

Fig. 2 Register mapping for the HP micro-kernel. The left-hand side matrix denotes the partial contribution to C_r computed at each iteration of loop L6 of the micro-kernel

full C_r via an outer product (rank-1 update) involving one column of A_r and one row of B_r. This column of A_r is stored in vectors $v_0 - v_2$ at even iterations and $v_3 - v_5$ at odd iterations; similarly, the row of B_r is stored in v_6 at even iterations and v_7 at odd iterations. Storing consecutive columns/rows in different vectors allows to overlap the computations of the "current" rank-1 update with the pre-fetching of the data necessary for the "next" rank-1 update. The rank-1 update is performed using 24 FMLA operations and only 2 SIMD loads. This renders a ratio of 12 floating point operations per load which favors performance. To further improve the performance of the micro-kernel we applied several additional fine-grain optimizations:

- Loop unrolling to reduce overhead due to loop control instructions.
- Data pre-fetching of A_r and B_r into the L1 cache inside loop L6 to hide memory transfer latency.
- Instruction re-ordering to take advantage of the processor instruction dispatch.

Algorithm 1 Optimized HP micro-kernel with micro-tiles $A_r \rightarrow 24 \times k_c$, $B_r \rightarrow k_c \times 8$, $C_r \rightarrow 24 \times 8$. For simplicity, loop unrolling and additional fine-grain optimizations are not displayed here. The actual implementation is done in ARM assembly code.

```
1: Pre-fetch $C_r$ to the L1 cache
2: Load column 0 of $A_r$ (LD1 3 registers over $v_0 - v_2$)
3: Load row 0 of $B_r$ (LD1 over $v_6$)
4: Clear $v_8 - v_{31}$ as accumulators for $C_r$
5: for $k_r = 0, 1, \ldots, k_c - 1$ do
6:     if (mod($k_r + 1, 2$) != 0) then
7:         Load column $k_r + 1$ of $A_r$ (LD1 3 registers over $v_3 - v_5$)
8:         Load row $k_r + 1$ of $B_r$ (LD1 over $v_7$)
9:         Multiply $v_0 - v_2$ with $v_6[0]$; accum. over $v_8 - v_{10}$ (3x FMLA[by element])
10:        Multiply $v_0 - v_2$ with $v_6[1]$; accum. over $v_{11} - v_{13}$ (3x FMLA[by element])
11:        Multiply $v_0 - v_2$ with $v_6[2]$; accum. over $v_{14} - v_{16}$ (3x FMLA[by element])
12:        Multiply $v_0 - v_2$ with $v_6[3]$; accum. over $v_{17} - v_{19}$ (3x FMLA[by element])
13:        Multiply $v_0 - v_2$ with $v_6[4]$; accum. over $v_{20} - v_{22}$ (3x FMLA[by element])
14:        Multiply $v_0 - v_2$ with $v_6[5]$; accum. over $v_{23} - v_{25}$ (3x FMLA[by element])
15:        Multiply $v_0 - v_2$ with $v_6[6]$; accum. over $v_{26} - v_{28}$ (3x FMLA[by element])
16:        Multiply $v_0 - v_2$ with $v_6[7]$; accum. over $v_{29} - v_{31}$ (3x FMLA[by element])
17:    else
18:        Load column $k_r + 1$ of $A_r$ (LD1 3 registers over $v_0 - v_2$)
19:        Load row $k_r + 1$ of $B_r$ (LD1 over $v_6$)
20:        Same code as in lines 9–16, with $v_0 - v_2$, $v_6[0]$ replaced by $v_3 - v_5$, $v_7[0]$, respectively
21:    end if
22: end for
23: Store $v_8 - v_{13}$, containing columns 0 and 1 of $C_r$ (2x ST1 3 registers)
24: Store $v_{14} - v_{25}$, containing columns 2 to 5 of $C_r$ (4x ST1 3 registers)
25: Store $v_{26} - v_{31}$, containing columns 6 and 7 of $C_r$ (2x ST1 3 registers)
```

– Instruction re-ordering taking into account instruction latencies in order to avoid processor stalls.

3.3 Selecting the cache configuration parameters

After developing the HP micro-kernel, the next step to obtain an optimized realisation of GEMM for the NVIDIA Carmel processor consists in determining the optimal values for the cache configuration parameters: m_c, n_c, k_c. Following the BLIS convention, we will focus on m_c and k_c, as these two parameters exert a much stronger impact on the performance of the operation, and we will fix n_c to the default value used in BLIS for other ARM processors: 3,072.

To carry out this search optimization, we first used the analytical model presented in [11] to identify the optimal values of k_c and m_c for the NVIDIA Carmel processor (see Sect. 4.1 for cache characteristics) and a GEMM kernel operating in HP/SP. When working with these precision formats, the analytical model recommends $m_c = 1,344$, $k_c = 684$ for HP, in comparison with $m_c = 896$, $k_c = 512$ for SP.

In addition, as the "m-dimension" of the matrix multiplication kernels that arise in many DL applications are usually much smaller than the optimal values for m_c determined by the analytical model, we performed an exhaustive experimental search

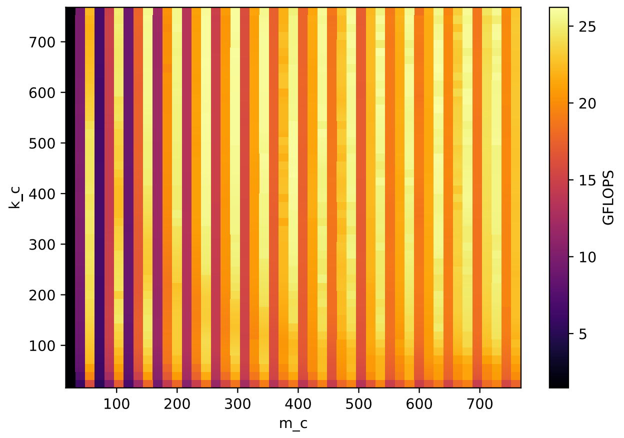

Fig. 3 Performance (in GFLOPS) of the HP realisation of GEMM on a single core of the NVIDIA Carmel processor for different values of m_c and k_c

covering the parameter space (m_c, k_c) for smaller sizes. Figure 3 shows a visualization of that parameter search for HP displaying the performance of the kernel measured in GFLOPS (billions of, in this case HP, floating point arithmetic operations per second). From that search, we obtained that $m_c = 576$, $k_c = 672$ offer fair performance for HP (26.65 GFLOPS using the analytical model parameters versus 26.22 GFLOPS with the values obtained via parameter search within the small problem range), and can be convenient choices for HP GEMM instances with small m-dimension. In comparison, a similar search for SP reported that $m_c = 560$, $k_c = 368$ is an appropriate pair for small SP GEMM kernels.

3.4 Packing routines

The BLIS implementation of GEMM includes two packing routines that rearrange the entries of the matrix operands A and B into specific layouts in buffers A_c and B_c, respectively. The packing is done to ensure that the contents of these buffers are located in the appropriate levels of the memory hierarchy and accessed with unit stride from the micro-kernel [8]. BLIS amortizes the cost of the packing with a large number of floating-point operations (flops): provided m, n, k are "large enough" and m_c, n_c, k_c are chosen optimally, the contribution of the packing routines to the cost of the global execution of GEMM is much lower than that of the micro-kernel. Nevertheless, the baseline realization of the packing routines in BLIS, for 32-bit and 64-bit operands, can be easily specialized for our HP scenario as follows:

- Development of 16-bit counterparts of the baseline packing routines in BLIS.
- Parallelization of the loops performing the matrix copies.

For the particular case of the convolution operator appearing in CNNs, the packing routines can be combined with the IM2COL operator in order to hide a considerable fraction of the memory and time costs of the latter. Concretely, this approach removes the need of the additional memory work space utilized by an explicit IM2COL transform, enabling efficient inference with large CNN models in platforms with limited memory capacity. Furthermore, the integration of the IM2COL operator inside GEMM (referred to hereafter as CONVGEMM) yields an efficient and portable implementation of the convolution operator [17].

3.5 Multi-threaded parallelization

For the execution with multiple cores, we leverage the OpenMP directive `#pragma omp parallel for` to distribute the iteration workspace of loop L4 among the NVIDIA Carmel processor cores. The parallelization of other loops in general offered lower performance for the target range of problem dimensions. The simultaneous parallelization of multiple loops (nested parallelism) is left as future work. In addition to that, the outermost loop of both packing routines is parallelized via the same OpenMP `#pragma` directive.

4 Experimental evaluation

In this section, we present several tests to assess the performance of our HP GEMM implementation. The experiments described next are split into two groups: standalone evaluation for the HP GEMM realization, and validation of the benefits when integrated into CONVGEMM.

4.1 Experimental configuration

The evaluation presented in this paper was executed on an NVIDIA Xavier platform, which embeds an NVIDIA Carmel octa-core CPU (implementing the ARMv8.2 micro-architecture), with the ARMv8.2-FP16 extension; an NVIDIA 512-CUDA core Volta GPU (not employed in the experiments); and 32 GB of main memory. On the software side, the experiments utilized the Ubuntu Linux distribution 18.04.4, the GNU C compiler gcc 10.0, and BLIS 0.7.0.

The NVIDIA Carmel processor consists of 4 core clusters each with 2 ARM v8.2 cores and 2MB 16-way set-associative of L2 data cache shared between them. Each core owns 64KB 4-way set-associative L1 data cache. All the core clusters are connected to the system fabric which provides coherence between the core clusters and the rest of the system. The coherence fabric also connects to a 4-MB 16-way set-associative victim-style L3 cache.

As our main interest is testing the HP realization in low-power systems, we set the *30W_all* power mode available in the *nvpmodel* power/performance management utility included in the Jetson Xavier. This mode activates all 8 cores and fixes their frequency to 1.2 GHz to ensure that the power consumption of the system stays within a 30-Watt budget. This also permits an easier evaluation of the multi-threaded parallelization, as it avoids side effects that can occur if the system automatically adjusts the core frequencies depending on the workload and number of active cores.

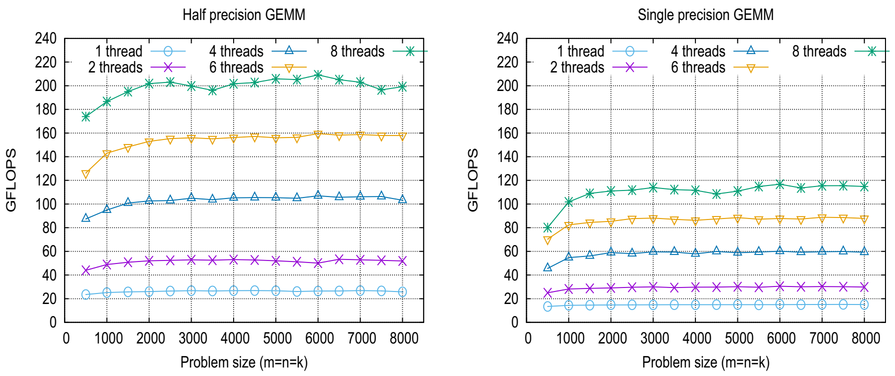

Fig. 4 Performance comparison between the two realizations: new HP GEMM (left) and the reference SP GEMM (right)

All tests were executed multiple times and the results were averaged to smooth out system load effects in the measurements.

4.2 Evaluation of the HP GEMM kernel

In order to test the performance of the new HP GEMM routine, we first compare this realization against the optimized SP GEMM in BLIS for ARMv8 processors using the m_c and k_c obtained by the parameter search for SP. We use those block sizes because BLIS does not include a configuration for the NVIDIA Carmel and the A-57 Configuration obtains suboptimal performance. We follow the convention of the BLIS performance benchmarks [13,18] and evaluate problems with equal matrix dimensions (that is, $m = n = k$). In addition, in order to assess the quality of the multi-threaded parallelization, we perform the evaluation for 1, 2, 4, 6, and 8 threads. Threading was applied to loop L4 and the packing routines, as described in Sect. 3.5.

Figure 4 compares the new HP GEMM and the reference SP GEMM for several problem sizes and threading configurations. The results there demonstrate that our HP GEMM achieves significant performance gains over the SP instance. For several problem sizes, the HP GEMM doubles the performance obtained by its SP counterpart, which corresponds to the maximum gain that could be expected due to the single-to-half precision reduction. In addition, the multi-threaded executions show that the selected parallelization scales properly with the number of cores.

4.3 Evaluation of the HP CONVGEMM kernel for CNNs

As part of our work, we also adapted the technique for the convolution operator in [17] to operate with our new HP GEMM kernel. In order to evaluate the performance of the resulting HP CONVGEMM kernel, we leveraged a CNN simulator to assess the impact of this change on the inference stage for several well-known DNNs. This simulator executes all the convolutional layers appearing in a specific CNN model via an integrated variant of the "IM2COL"-approach [7,17], for a certain precision and batch

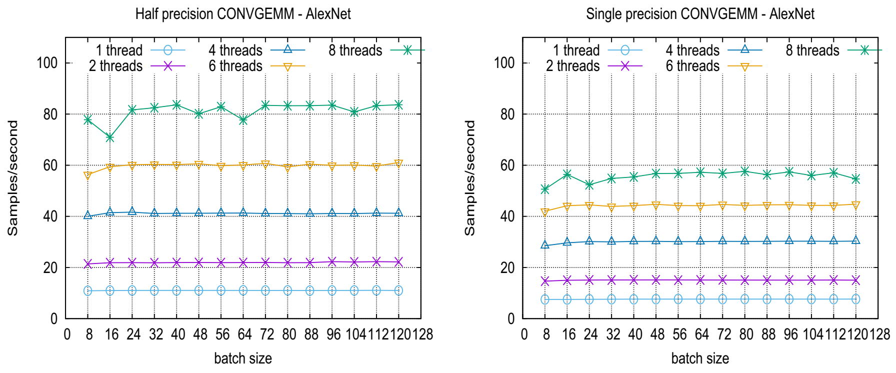

Fig. 5 Performance comparison between the HP and SP CONVGEMM algorithm (left and right, resp.) for AlexNet

size (that is, number of simultaneous input samples to the CNN). At this point, we recognize that there exist other alternatives to realize the convolution operator —based, for instance, on the direct convolution, the fast Fourier transform, or the Winograd algorithm; see, e.g., the references in [17]— that may deliver higher performance depending on the dimensions of the convolution operands. Therefore, casting all the layers of a CNN as IM2COL-based operators will likely yield a suboptimal solution.

Figure 5 compares the performance obtained by the two CONVGEMM algorithms, respectively, using HP and SP, when integrated into the CNN simulator. The throughput results there are measured in terms of processed samples per second during the CNN inference stage, and comprise a variety of different batch dimensions and number of threads. The figure shows that the new HP convolution outperforms the SP variant, delivering fair throughput gains for the inference stage.

A closer inspection reveals that the performance improvements offered by our HP CONVGEMM realization are lower than the gains that were exposed for the HP GEMM kernel in the previous subsection. This behavior is due to the reduced dimensions of several of the GEMM kernels arising in the selected CNNs. To illustrate this in detail, let us consider the first layer in AlexNet. In this particular case, the application of the IM2COL transform yields a GEMM kernel with dimensions $m = 64, k = 363$ (and very large n). Now, if we compare these values against the experimental quasi-optimal configuration parameters for HP and SP ($m_c, k_c = 576, 672$ for HP and 560, 368 for SP), we can conclude that, in both cases, there may appear a suboptimal utilization of the cache memories which does not amortize the cost of transferring the data across the memory hierarchy with enough floating-point operations.

For example, with the experimental optimal parameters for HP, the buffer A_c (of dimension $m_c \times k_c$) occupies 738.3 KB out of the 2 MB L2 cache per core of the NVIDIA Carmel processor(36%) and a similar amount for SP (786.1 KB, 38.3%). However, due to the actual dimensions of the GEMM kernel involved in the first convolutional layer of AlexNet, $m_c = m = 64$ and $k_c = k = 363$. Therefore, A_c only consumes 44.3 KB for HP (2.2% of the L2 cache) and twice as much for SP.

A similar (or even worse) problem occurs for the re-utilization of B_r (of dimension $k_c \times n_r$, with $n_r = 8/12$ for the HP/SP micro-kernel) in the L1 cache. With the optimal configuration parameters, in the HP realization B_r consumes 10.5 KB (16.4% of the 64-KB L1 cache); and in the SP realization, 17.2 KB (26.9%). In contrast, for the first layer of AlexNet, the HP values are reduced to 5.7 KB (8.9%) but remain at 17.1 KB (26.7%) for SP. This shows a better utilization of the L1 cache by the SP micro-kernel, partially due to the adoption of a larger value for the micro-kernel parameter n_r.

An additional performance hazard arises for small GEMM kernels when the m-dimension of the operand is not an integer multiple of m_r. In those cases, the optimized micro-kernel is used $\lfloor m/m_r \rfloor$ times out of each $\lceil m/m_r \rceil$ invocations but, for the remaining calls, a baseline, less optimized C code needs to be employed. For the first layer of AlexNet, for example, with $m = 64$ and $m_r = 24$, the optimized micro-kernel is invoked 2/3 of the times (processing 24 "m-rows" of the problem each) and the baseline one in 1/3 of the cases (for the remaining 16 m-rows). This has a strong negative impact on performance.

To tackle this problem, we have developed a second micro-kernel with $m_r \times n_r = 8 \times 8$. This complementary micro-kernel does not exhibit such a high utilization of the processor registers, and therefore delivers a lower performance than the $m_r \times n_r = 24 \times 8$ one. On the other hand, the smaller value of m_r is a better match for GEMM kernels with small m-dimension, as it reduces the amount of work cast in terms of the baseline micro-kernel. Finally, we have combined both micro-kernels, by merging them into a single code and adjusting the packing routines: Consider for example a GEMM with $m = 65$. Our dual-μkernel GEMM instance employs the 24×8 micro-kernel to process the top $24 + 24$ m-rows; the 8×8 micro-kernel to process the middle $8 + 8$ m-rows, and the baseline code to process the bottom m-row.

5 Concluding remarks

We have presented an architecture-specific realization of the GEMM kernel optimized for the NVIDIA Carmel processor (ARMv8.2 architecture) and HP floating point arithmetic by adapting the generic instance of this kernel in the BLIS library. The new kernel automatically carries over to a major fraction of the BLAS functionality implemented in BLIS.

From the point of view of performance, the HP GEMM kernel basically doubles the GFLOPS rates attained by the optimized SP GEMM included in BLIS for the ARMv8 architecture. On the one hand, this is especially relevant for DL transformers (of wide appeal for NLP tasks), as the training and inference stages for this type of DNNs usually involve large GEMM kernels. On the other hand, the HP GEMM kernel can be combined with our CONVGEMM algorithm for CNNs and, while the performance gains are more modest in this case (due to the restricted dimensions of the GEMM kernels), the resulting HP convolution operator still offers a significant gain against its SP counterpart.

As an important additional advantage in deep learning, using HP instead of SP saves half of the memory which would be necessary to store the model parameters and activations of a CNN in SP. This in turn allows the in-core evaluation of larger models and/or reduces the pressure on the hardware memory capacity.

Acknowledgements This work was supported by projects TIN2017-82972-R and RTI2018-093684-B-I00 from the *Ministerio de Ciencia, Innovación y Universidades*, project S2018/TCS-4423 of the *Comunidad de Madrid*, project PR65/19-22445 of the *UCM*, and project Prometeo/2019/109 of the *Generalitat Valenciana*.

References

1. Deng L et al (2013) Recent advances in deep learning for speech research at Microsoft. In: 2013 IEEE international conference on acoustics, speech and signal processing, May, pp 8604–8608
2. Krizhevsky A, Sutskever I, Hinton GE (2012) ImageNet classification with deep convolutional neural networks. In: Proceedings of the 25th international conference on neural information processing systems—vol 1, ser. NIPS'12. Curran Associates Inc., USA, pp 1097–1105
3. Zhang J, Zong C (2015) Deep neural networks in machine translation: an overview. IEEE Intell Syst 30(5):16–25
4. Devlin J et al (2019) BERT: pre-training of deep bidirectional transformers for language understanding. In: Proceedings of 2019 conference North American Chapter of the Association for Computational Linguistics: Human Language Technologies, vol 1, pp 4171–4186
5. Sze V et al (2017) Efficient processing of deep neural networks: a tutorial and survey. Proc IEEE 105(12):2295–2329
6. Vaswani A et al (2017) Attention is all you need. In: Advances in neural information processing systems, vol 30, pp 5998–6008
7. Chellapilla K, Puri S, Simard P (2006) High performance convolutional neural networks for document processing. In: International workshop on frontiers in handwriting recognition, available as INRIA-00112631 report from https://hal.inria.fr/inria-00112631
8. Van Zee FG, van de Geijn RA (2015) BLIS: a framework for rapidly instantiating BLAS functionality. ACM Trans Math Softw 41(3):14:1–14:33
9. Dongarra JJ, Du Croz J, Hammarling S, Duff I (1990) A set of level 3 basic linear algebra subprograms. ACM Trans Math Softw 16(1):1–17
10. Goto K, van de Geijn R (2008) Anatomy of high-performance matrix multiplication. ACM Trans Math Softw 34(3):12:1–12:25
11. Low TM, Igual FD, Smith TM, Quintana-Orti ES (2016) Analytical modeling is enough for high-performance blis. ACM Trans Math Softw 43(2):1–18. https://doi.org/10.1145/2925987
12. Fabeiro JF, Andrade D, Fraguela BB (2016) Writing a performance-portable matrix multiplication. Parallel Comput 52:65–77
13. Zee FGV, Smith TM, Marker B, Low TM, Geijn RAVD, Igual FD, Smelyanskiy M, Zhang X, Kistler M, Austel V, Gunnels JA, Killough L (2016) The BLIS framework: experiments in portability. ACM Trans Math Softw 42(2):1–19. https://doi.org/10.1145/2755561
14. Smith TM, van de Geijn R, Smelyanskiy M, Hammond JR, Zee FGV (2014) Anatomy of high-performance many-threaded matrix multiplication. In: IPDPS '14: Proceedings of the international parallel and distributed processing symposium (to appear)
15. Catalán S et al (2016) Architecture-aware configuration and scheduling of matrix multiplication on asymmetric multicore processors. Cluster Comput 19(3):1037–1051
16. Hennessy JL, Patterson DA (2003) Computer architecture: a quantitative approach. Morgan Kaufmann Pub, San Francisco
17. San Juan P, Castelló PS, Dolz MF, Alonso-Jordá P, Quintana-Ortí ES (2020) High performance and portable convolution operators for multicore processors. In: Proceedings of 32nd international Symposium on Computer Architecture and High Performance Computing (SBAC-PAD), pp 91–98
18. BLIS Performance benchmarks (2020). https://github.com/flame/blis/blob/master/docs/Performance.md