Scalable and Efficient Linear Algebra Kernel Mapping for Low Energy Consumption on the *Layers* CGRA

Zoltán Endre Rákossy[1(✉)], Dominik Stengele[2], Axel Acosta-Aponte[1], Saumitra Chafekar[1], Paolo Bientinesi[2], and Anupam Chattopadhyay[3]

[1] Institute for Communication Technologies and Embedded Systems (ICE), Aachen, Germany
rakossy@ice.rwth-aachen.de
[2] Algorithmically-Driven Code Generation for High-Performance, Computing Architectures, AICES, RWTH Aachen University, Aachen, Germany
[3] School of Computer Engineering, Nanyang Technological University, Nanyang, Singapore

Abstract. A scalable mapping is proposed for 3 important kernels from the Numerical Linear Algebra domain, to exploit architectural features to reach asymptotically optimal efficiency and a low energy consumption. Performance and power evaluations were done with input data set matrix sizes ranging from 64×64 to 16384×16384. 12 architectural variants with up to 10×10 processing elements were used to explore scalability of the mapping and the architecture, achieving $< 10\,\%$ energy increase for architectures up to 8×8 PEs coupled with performance speed-ups of more than an order of magnitude. This enables a clean area-performance trade-off on the *Layers* architecture while keeping energy constant over the variants.

1 Introduction

Accelerating applications using coarse-grained reconfigurable architectures (CGRA) promises great benefits [2,3], but without an efficient compiler, designers are forced to go over tedious manual application mapping processes to find and leverage maximum performance with minimum energy. These must be repeated if architectural or application parameters are changed, therefore mapping *scalability* is an important factor.

In this paper, we propose scalable and highly efficient mapping solutions for 3 Numerical Linear Algebra kernels and apply them on a CGRA designed for scalability, called *Layers* [9]. Computationally demanding and highly parallel, NLA kernels represent the perfect application domain to fully tap the advantages of CGRA parallel execution and flexibility. However, high parallelism also requires high storage access pressure in various patterns, which is one of the limiting factors when seeking efficient execution on CGRAs. Many times, optimal execution is limited by storage bandwidth, or the hardware processing resources do not fit

K. Sano et al. (Eds.): ARC 2015, LNCS 9040, pp. 301–310, 2015.
DOI: 10.1007/978-3-319-16214-0_25

the optimal algorithmic execution window. We discuss how the complex interplay between application mapping, scaling and architectural features influence energy efficiency.

2 The *Layers* Architecture

The Layers architecture is a new multi-layered scalable and parameterizable CGRA[9], developed initially as a 4×4 fixed architecture[10], using a high-level design methodology proposed in [8]. The core philosophy aligns with the functional separation into layers of control (Q), computation (L0), communication (L1) and memory access (L2) with dedicated hardware structures for each of task class, maximizing parallelism and computational efficiency to achieve low energy. The architecture is organized in a 3D pipeline structure (Fig. 1), where data flows from the memory banks via the memory and communication layers to the computation layer in a vertical pipeline, while control flow is processed in the horizontal pipeline. Conceptually, the architecture represents for each of the task classes, a set of elementary hardware structures, which can be grouped into higher order functions by means of reconfiguration. The application data/control path is thus reconstructed out of small pieces, allowing to be arranged differently when the application changes. Rearranging the existing elementary structures is done via functional calls from assembly, in a data-independent way, for describing the application kernel. Each layer can work at a ratio $r = 2^n$ vs. the main pipeline, balancing task class bottle-necks. For each kernel, the set of function calls (i.e. configuration contexts) is stored in the program memory, as a self-contained black-box procedure derived from the mapping. An interested reader is invited to read further architectural details in the references mentioned above. For understanding the proposed mapping solutions in this paper, however, it is sufficient to keep the following key consideration in mind, when targeting *energy efficiency*: all floating-point PEs in the computation layer (L0) must be busy doing meaningful work at all times to exploit maximum parallelism. Furthermore, this consideration must be true, even if the number of processing elements and other architectural parameters change, hence he mapping must be *scalable*.

3 Mapping the Algorithms

3.1 Preliminaries

In this paper we focus on three kernels: dot product (DOT), matrix-vector (GEMV) and matrix-matrix multiplication (GEMM). The derivation of the mapping took into consideration several architectural and algorithmic parameters, while trying to keep it as generic as possible. Column-based memory loads have been avoided due to creating single-cycle access conflicts on modulo-P based memory bank distributions, common for CGRAs. Efficiency evaluation is performed by taking the ratio between the theoretically required number of execution cycles c_{kmin} and the number of actual cycles executed by the architecture

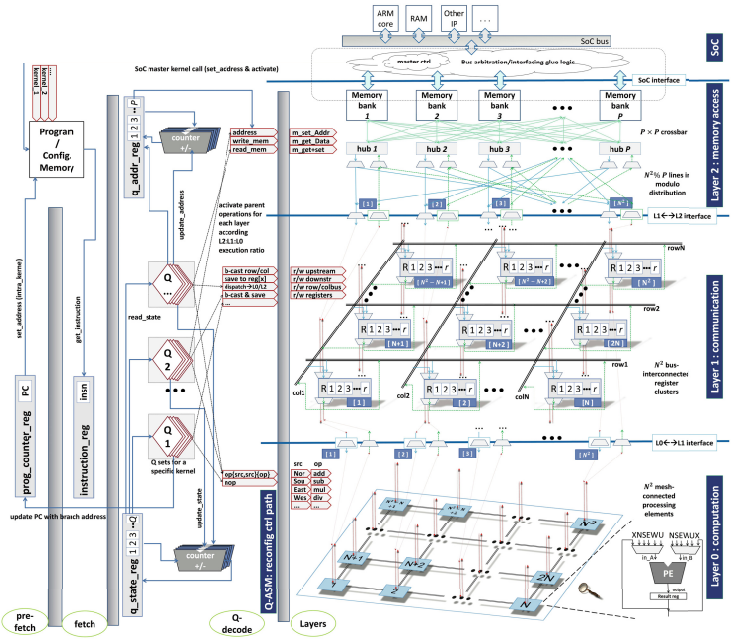
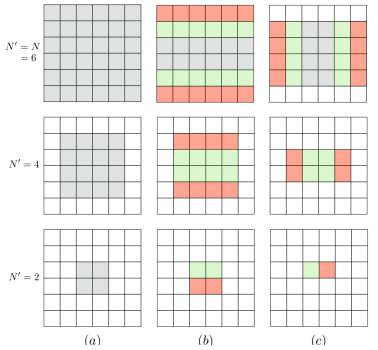


Fig. 1. The *Layers* architecture: scalable and modular layers dedicated for memory access, communication and computation are managed by a reconfigurable control flow stage.

c_k, for each problem size N, yielding $\eta_k(\{\cdot\}, N) := \frac{c_{k\,min}(\{\cdot\}, N)}{c_k(\{\cdot\}, N)}$. Theoretical values are derived from all arithmetic operations for a kernel, taking into account hardware parameters, e.g. FP division takes 4 cycles, while all other operations take 1 cycle. This allows evaluating mapping performance directly. Although architectural parameters specific to *Layers* are used here, please note that the mapping is valid for any mesh-connected architecture of processing elements, as long as the necessary data can be provided for the elements at the required time.

Generally, when mapping in a *scalable* way on a scalable architecture, the execution window size has to match the size of the array for efficiency and respect available memory bandwidth. An efficient block-based scheduling and mapping solution is discussed in earlier work[10], where *Layers* had fixed 4×4 PEs and 8 ports, yielding a fixed mapping, while automation of this is attempted in [4]. Here we show that a manual mapping can be derived for scalability yielding great performance and energy values when scaling. Further complexity arises when the array size modulo matrix size does not cleanly match at the end of the data. *Layers* uses `override` signals, set by the control layer (Q), to disable the extra execution units when required.




```
N′ ← N
while N′ > 3 do
    // 1st L0-cycle
    for each element in the second row of the current square do
        ADD(X, N)
    end for
    for each element in the second last row of the current square do
        ADD(X, S)
    end for
    // 2nd L0-cycle
    for each element in the second column of the current square do
        ADD(X, W)
    end for
    for each element in the second last column of the current square do
        ADD(X, E)
    end for
    // Update N′
    N′ ← N′ − 2
end while
```

Fig. 2. DOT algorithm with the accumulation procedure (folding). (a) initial state, (b) and (c) show the iteration transition, which reduces to the same problem of smaller size. Repeating such iterations will reduce to a 2×2 or 3×3 special case, which finally yield the final result in one element.

3.2 Accumulation Folding Procedure

Mapping: Although not a kernel *per se*, most kernels make use of this procedure, especially in the epilogue portion of hot-spot loops. Fig. 2 shows how partial sums can be folded into the final value for a 6×6 architecture, following a generic algorithm valid for any square array size. To preserve simplicity and scalability, the algorithm starts from the edge of the array, horizontally or vertically, creating two addition fronts. The procedure is repeated until the folding front reaches a 2×2 or a 3×3 data square. $N' = 2$ and $N' = 3$ have to be treated differently, because it is not possible anymore to reduce the square from top/bottom or left/right simultaneously. For these cases, only `ADD(self, south)` and `ADD(self, east)` are inserted, reducing $N' = 2$ to a single element, or, if N is odd, $N' = 3$ to $N' = 2$ and to a single element afterwards.

Complexity: $op_{acc}(N) = N^2 - 1$ because always $i - 1$ additions are required to add up i values, only depending on the architecture size N. When N is even, the square will be reduced to a single element in $N/2$ iterations of the accumulation procedure. Every iteration consists of 2 *L0-cycles*, hence it will take $8N$ *cycles*, for $r_{L0:L1:L2} = 1 : 8 : 8$. When N is odd, we cannot reduce a square with $N' = 3$ to a single element in just one iteration. Hence, we need $(N + 1)/2$ iterations, which results in $8(N + 1)$ *cycles*.

3.3 Dot Product (DOT): $c = \sum_{i=1}^{n} a_i \cdot b_i$

Mapping: When mapping in a scalable way, the execution window size has to match the size of the array. Fig. 3 shows how each element of a $N^2 = 4$ element array is assigned an operation within the execution window, which partitions the two vertical data columns. If N^2 changes, the execution window scales accordingly. After an initial multiplication on all elements, the execution window

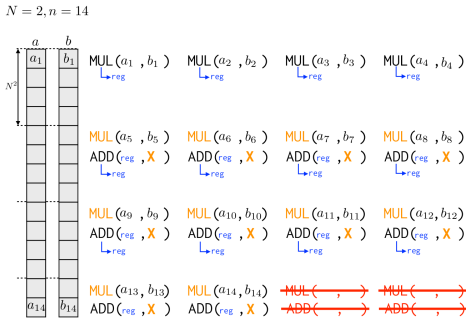


Fig. 3. DOT mapping with N=2, yielding an execution window of 4 elements, which slides downwards on the two data columns a, b, executing multiply and accumulate instructions alternately, until end of data is reached. If data size does not match N^2, `overrides` deactivate extra instructions. X denotes taking previous output (self).

slides downwards through the data, multiplying and accumulating the results for each processing element. Partial accumulation results are sent to L1 registers for the duration of one L0-cycle and used again in subsequent accumulation cycles, avoiding storing back to memory. When the $N^2 \% n$ does not cleanly match the array at the end of the data, `overrides` are set by the control core, disabling the extra execution units which have no data. This is a source of efficiency loss, however for very large matrices, the amount of full execution windows is dominating. After moving the execution window in N^2 steps, every element holds one partial result and the final result c is the sum of all of those, hence by calling the accumulation folding procedure the kernel is completed.

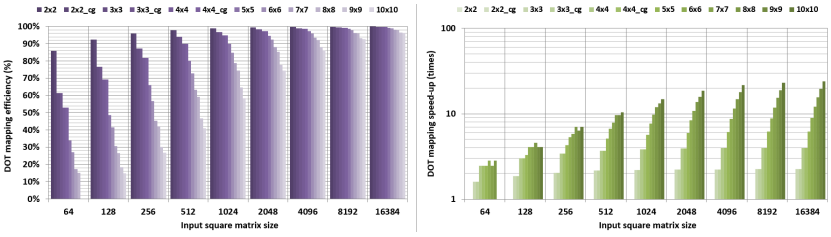


Fig. 4. DOT mapping efficiency and mapping-based speed-up for various architectures (N=2..8) and data sizes (64..16384). Due to the accumulation procedure, efficiency receives a heavy penalty, especially for large arrays with small input data sizes. When enough data is used, the penalty is much smaller. Worst penalty for large data sets on large arrays is $< 4\%$. This has the worst efficiency of the 3 kernels.

Complexity: For vectors of length n, n multiplications and $n-1$ additions are required, hence the operation complexity is $op_{dot}(\{n\}) = 2n - 1$. Hence, $c_{\texttt{dot}\,min}(\{n\}, N) = \frac{8}{N^2}(2n-1)$.

Efficiency: Mapping efficiency is shown in Fig. 4 for different architecture sizes and input vector lengths, where inefficiencies of the accumulation procedure are dominating large architectures on small data sets. The expected speedup from this mapping when scaling N, is shown on the right side of Fig. 4, closing within $< 4\%$ to the expected theoretical speed-up value for large data sets on large arrays.

3.4 General Matrix-Vector Multiplication (GEMV) $c_i = \sum_{j=1}^{m} a_{ij} \cdot b_j$

Mapping: It is immediately visible from the formula that every multiplication can be done in parallel, but in the end every product has to be added together for every row i. This breaks the symmetry of the algorithmic progress through the data. It would be beneficial if the computation of $c_1, c_2, \ldots, c_{N^2}$ can be assigned to L0-elements $e_0, e_1, \ldots, e_{N^2-1}$. This would yield N^2 elements of c after m multiplications and $m-1$ additions. However, while any L0-element e_{i-1} would operate exclusively on row i of A, the input data a_{ij} in every step would make loading values of a column of A necessary, breaking scalability and efficiency for large N.

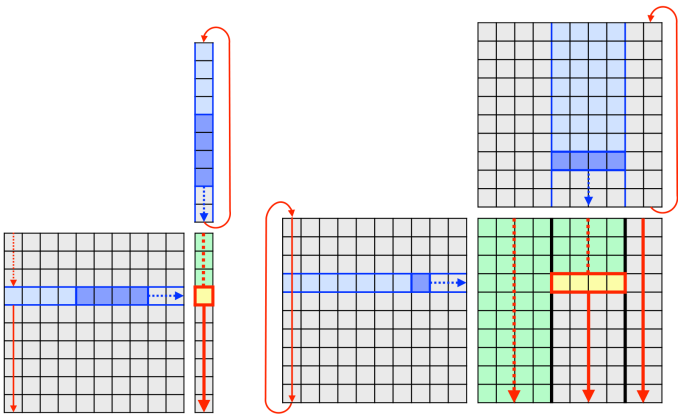


Fig. 5. GEMV(left) and GEMM(right) scalable execution window progress (here, $N^2 = 4$ elements), denoted in dark blue, yielding the result in yellow. The progression of this window through the data ensures no more than one load per memory port per cycle.

To operate on rows only, at any given point in time, a less optimal but scalable scheduling was implemented shown in Fig. 5(left), thus N^2 L0-elements can work in parallel to calculate every c_i. The execution window moves in N^2 steps horizontally in the matrix and vertically on the vector, but forces an accumulation procedure at the row boundary of each row.

Complexity: Operation complexity of GEMV, with input size n and m, is equivalent to n times DOT for input size m, hence $op_{gemv}(\{n, m\}) = n \cdot op_{\texttt{dot}}(\{m\}) = n(2m - 1)$. Therefore, expected minimum on *Layers* is $c_{\texttt{gemv}_{min}}(\{n, m\}, N) = \frac{8}{N^2} n(2m - 1)$.

Efficiency: Mapping efficiency is similar to, but better than the DOT product efficiency shown in Fig. 4, due to the extra accumulation procedures for every row. For large arrays and large data sets is $< 3\%$ close to the expected theoretical maximum.

3.5 General Matrix Multiplication (GEMM) $c_{ij} = \sum_{l=1}^{k} a_{il} \cdot b_{lj}$.

Mapping: Respecting the same constraint of not loading data on the column (keeping one access per memory port), for *GEMM* the data dependencies turn out to be problematic. Most obvious mapping solutions would require to load a column of values from either A or B and multiply it with a row from the other. Rectangular or square windows with a height of more than 1 were not possible either since this would require to work on columns in either matrix. Block-based approaches are not scalable when modifying N or memory port amount P and produce complex addressing problems for a manual mapping. To bypass this, the following mapping is proposed, allowing a scalable execution window without column loads, as shown in Fig. 5(right): 1) Load a_{11} 2) Multiply a_{11} with $b_{11}, \ldots, b_{1N^2}$ 3) Continue with a_{12} and $b_{21}, \ldots, b_{2N^2}$ and accumulate the partial results 4) When finishing with the last row, store the resulting $c_{11}, \ldots, c_{1N^2}$ and continue with the next window. This implies more loads due to reiterating through the data several times, but scales perfectly since a window of height 1 and width N^2 can be used, without causing memory port conflict or exceeding available bandwidth. The window iterates matrix B and C column-wise and only in the last column of windows there may be overrides necessary, making the override logic efficient.

Complexity: *GEMM*, with input size n, m and k, is equivalent to m times *GEMV* for input size n and k, hence $op_{\text{gemm}}(\{n, m, k\}) = m \cdot op_{\text{gemv}}(\{n, k\}) = mn(2k-1)$. Hence, $c_{\text{gemm}_{min}}(\{n, m, k\}, N) = \frac{8}{N^2}\, mn(2k-1)$.

Efficiency: Even with the constraints considered, the implementation of *GEMM* is very efficient because the actual core of the implementation consists of 2 *L0-cycles* only, in which all processing elements are always occupied. For scaling by x amount of elements a speedup of close to x is achieved. Efficiency reaches optimality when the data set is a multiple of N^2 for large arrays.

4 Evaluation and Results

4.1 General Considerations

Layers has been coded completely in the LISA ADL of Synopsys Processor Designer, completely parametrized for easy scalability. Simulations have been conducted for random square input matrices of size 64..16384, for different combinations of $P = 2..32$ and $N = 2..10$. The assembly programs for each algorithm have been manually coded in a scalable way (auto-generation via embedded Ruby code), 5 L0-cycles for GEMM, 13 L0-cycles for GEMV, 11 L0-cycles for DOT including addition folding. Values are for single-precision floating point (32-bit).

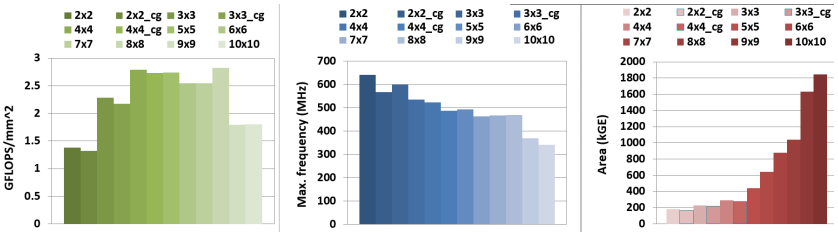


Fig. 6. Area, frequency and performance density for the Layers architecture

For these configurations RTL code has been generated and synthesized with DC I-2013 for Faraday 65nm technology library, using PowerCompiler and backward switching activity files for power estimation.

4.2 Time and Energy

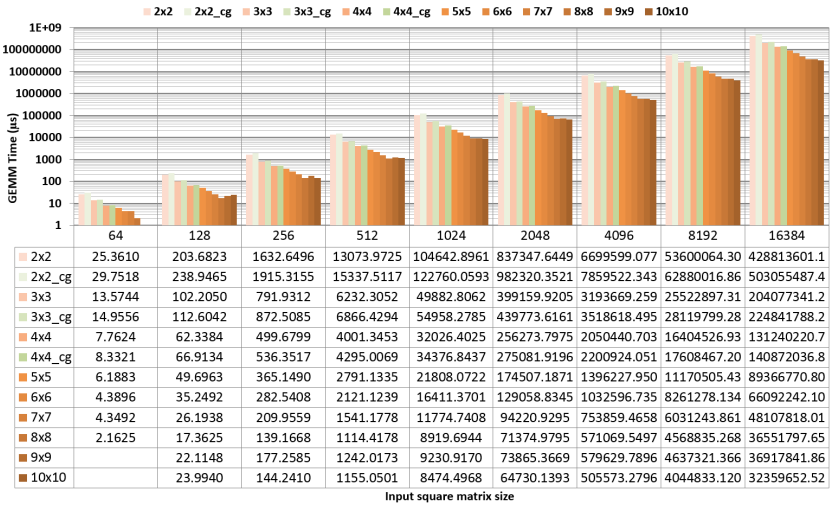
Very interesting results are provided in Fig. 7 for GEMM kernel, where the overall time and energy values are depicted for each configuration and input matrix size. Similar results are available for DOT and GEMV. Execution time spreads over several orders of magnitude with varying input data size, while an order of magnitude speed-up can be maintained between the smallest and largest array for large input data sizes. Except for the largest architectures, where the critical path of the L1 structures severely affected frequency and thus energy, the architecture and mapping scale with almost constant energy (<10% variance), translating into a clean trade-off between area and speed, without affecting energy. Clock-gating optimizations improved results for smaller designs by roughly 20-40%. Power values range between 13.71 mW for the 2×2 clock-gated variant to 418 mW for the 10×10 array. The 3×3 clock-gated variant reaches highest average efficiency with 23.48 GFLOPs/W, while the 10×10 variant reaches lowest at 10.17 GFLOPs/W. Reducing the interconnect length in L1 would cancel out the frequency penalty for the 2 largest designs, and thus the extra energy compared to the other designs.

4.3 Comparisons with Related Work

Fig. 6 provides some area, frequency and performance density data. The frequency of each architecture is limited by the control flow complexity in the q-decode stage for small N, and by L1 critical path for larger N at $r = 1 : 8 : 8$.

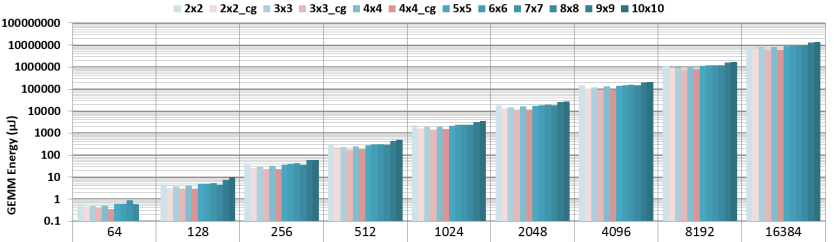
Unfortunately we found no works in the literature which map these kernels on a CGRA and provide detailed energy results, allowing only coarse comparisons in terms of overall power efficiency or power density [9]. Only aggregate results could be found also for GPGPU solutions[11] and a DSP[1]. In [5], authors presented a novel FPGA-based fine-grained reconfigurable architecture to map

several numerical linear algebra kernels and compared with Intel Xeon Woodcrest processor to report 10-150× speed-up/energy-efficiency improvement. However, no absolute results in time or energy for any particular technology node is reported making it extremely difficult to compare with our proposed approach. A more detailed implementation for our target kernels are reported in [6], where the total performance results include the communication bandwidth with a PC. Considering the overall performance, our implementation is clearly superior by several orders of magnitude, though, the comparison is not accu-



	64	128	256	512	1024	2048	4096	8192	16384
2x2	25.3610	203.6823	1632.6496	13073.9725	104642.8961	837347.6449	6699599.077	53600064.30	428813601.1
2x2_cg	29.7518	238.9465	1915.3155	15337.5117	122760.0593	982320.3521	7859522.343	62880016.86	503055487.4
3x3	13.5744	102.2050	791.9312	6232.3052	49882.8062	399159.9205	3193669.259	25522897.31	204077341.2
3x3_cg	14.9556	112.6042	872.5085	6866.4294	54958.2785	439773.6161	3518618.495	28119799.28	224841788.2
4x4	7.7624	62.3384	499.6799	4001.3453	32026.4025	256273.7975	2050440.703	16404526.93	131240220.7
4x4_cg	8.3321	66.9134	536.3517	4295.0069	34376.8437	275081.9196	2200924.051	17608467.20	140872036.8
5x5	6.1883	49.6963	365.1490	2791.1335	21808.0722	174507.1871	1396227.950	11170505.43	89366770.80
6x6	4.3896	35.2492	282.5408	2121.1239	16411.3701	129058.8345	1032596.735	8261278.134	66092242.10
7x7	4.3492	26.1938	209.9559	1541.1778	11774.7408	94220.9295	753859.4658	6031243.861	48107818.01
8x8	2.1625	17.3625	139.1668	1114.4178	8919.6944	71374.9795	571069.5497	4568835.268	36551797.65
9x9		22.1148	177.2585	1242.0173	9230.9170	73865.3669	579629.7896	4637321.366	36917841.86
10x10		23.9940	144.2410	1155.0501	8474.4968	64730.1393	505573.2796	4044833.120	32359652.52

Input square matrix size



	64	128	256	512	1024	2048	4096	8192	16384
2x2	0.5516	4.4301	35.5100	284.3576	2275.9725	18212.2275	145715.6100	1165796.038	9326652.943
2x2_cg	0.3922	3.1502	25.2508	202.2037	1618.4204	12950.5254	103616.8530	828985.4242	6632085.794
3x3	0.5084	3.8278	29.6599	233.4163	1868.2433	14949.5968	119611.3770	955900.1401	7643237.234
3x3_cg	0.3894	2.9317	22.7163	178.7720	1430.8745	11449.7919	91609.5196	732117.2523	5853902.105
4x4	0.4918	3.9497	31.6595	253.5232	2029.1769	16237.3797	129914.8978	1039382.624	8315314.766
4x4_cg	0.3603	2.8938	23.1959	185.7487	1486.7160	11896.6328	95184.6830	761523.9465	6092377.501
5x5	0.6196	4.9756	36.5585	279.4470	2183.4144	17471.5810	139789.7141	1118385.977	8947360.877
6x6	0.6294	5.0544	40.5138	304.1501	2353.2428	18505.8753	148065.0785	1184592.933	9477032.688
7x7	0.8860	5.3363	42.7727	313.9722	2398.7767	19194.8998	153577.9466	1228698.569	9800632.929
8x8	0.5674	4.5556	36.5147	292.4023	2340.3601	18727.4528	149837.9137	1198776.480	9590504.530
9x9		7.5836	60.7855	425.9129	3165.4684	25329.9301	198766.7924	1590231.402	12659875.56
10x10		10.0290	60.2898	482.7871	3542.1651	27055.8648	211319.2160	1690656.920	13525668.14

Input square matrix size

Fig. 7. Timing and energy results for GEMM. Clock-gated designs (*_cg) perform better. GEMV and DOT show similar trends. Constant energy is required for the same problem size across architectures, giving a clean performance:area trade-off.

rate as we measured the performance of a stand-alone core without considering complete system integration and communication latency with a host CPU. No clear way of separating the performance contributors could be extracted from LAC CGRA[7] either, which shows impressive aggregate results extracted from estimations without actual post-synthesis energy consumption data.

5 Conclusions

The proposed mapping solutions for efficient execution on CGRA-like architectures architecture reach close to 100% mapping efficiency and highlight the advantages of scalable algorithm-hardware co-design over 12 architectural variants of the *Layers* architecture. Constant energy when trading off area and performance is made possible.

References

1. Ali, M., Stotzer, E., Igual, F.D., van de Geijn, R.A.: Level-3 BLAS on the TI C6678 multi-core DSP. In: Proc. of the 2012 IEEE 24th Intl. Simp. on Computer Architecture and High Performance Computing (SBAC-PAD), pp. 179–186. IEEE (2012)
2. Chattopadhyay, A.: Ingredients of adaptability: a survey of reconfigurable processors. VLSI Design **2013**, 10 (2013)
3. DeHon, A.: The density advantage of configurable computing. Computer **33**(4), 41–49 (2000)
4. Fell, A., Rákossy, Z.E., Chattopadhyay, A.: Force-directed scheduling for data-flow graph mapping on coarse-grained reconfigurable architectures. In: Reconfigurable Computing and FPGAs (ReConFig), IEEE (2014)
5. Gonzalez, J., Núñez, R.C.: LAPACKrc: Fast linear algebra kernels/solvers for FPGA accelerators. In: Journal of Physics: Conference Series **180**, p. 012042. IOP Publishing (2009)
6. Lei, Y., Dou, Y., Dong, Y., Zhou, J., Xia, F.: FPGA implementation of an exact dot product and its application in variable-precision floating-point arithmetic. The Journal of Supercomputing **64**(2), 580–605 (2013). http://dx.doi.org/10.1007/s11227-012-0860-0
7. Pedram, A., van de Geijn, R.A., Gerstlauer, A.: Codesign tradeoffs for high-performance, low-power linear algebra architectures. IEEE Trans. Comput. **61**(12), 1724–1736 (2012)
8. Rákossy, Z.E., Acosta Aponte, A., Chattopadhyay, A.: Exploiting architecture description language for diverse IP synthesis in heterogeneous MPSoC. In: Reconfigurable Computing and FPGAs (ReConFig). IEEE (2013)
9. Rákossy, Z.E., Merchant, F., Acosta Aponte, A., Nandy, S., Chattopadhyay, A.: Scalable and energy-efficient reconfigurable accelerator for column-wise givens rotation. In: 22nd International Conference on Very Large Scale Integration (VLSI-SoC). IEEE (2014)
10. Rákossy, Z.E., Naphade, T., Chattopadhyay, A.: Design and analysis of layered coarse-grained reconfigurable architecture. In: Reconfigurable Computing and FPGAs (ReConFig), pp. 1–6 (2012)
11. Volkov, V., Demmel, J.W.: Benchmarking gpus to tune dense linear algebra. In: Proc. of the 2008 ACM/IEEE Conf. on Supercomputing, p. 31. IEEE Press (2008)