

Floating-Point Matrix Multiplication in a Polymorphic Processor

Georgi Kuzmanov, *Member, IEEE* and Wouter M. van Oijen

Abstract—We consider 64-bit floating-point matrix multiplication in the context of polymorphic processor architectures. Our proposal provides a complete and performance efficient solution of the matrix multiplication problem, including hardware design and software interface. We adopt previous ideas¹, originally proposed for loosely coupled processors and message passing communications. We employ these ideas into a tightly coupled custom computing unit (CCU) in the Molen polymorphic processor. Furthermore, we introduce a controller, which facilitates the efficient operation of the multiplier processing elements (PEs) in a polymorphic environment. The design is evaluated theoretically and through real hardware experiments. More precisely, we fit 9 processing elements in an XC2VP30–6 device; this configuration suggests theoretical peak performance of 1.80 GFLOPS. In practice, we measured sustained performance of up to 1.79 GFLOPS for the matrix multiplication on real hardware, including the software overhead. Theoretical analysis and experimental results suggest that the design efficiency scales better for large problem sizes.

Index Terms—Floating-point arithmetic, Matrix multiplication, Polymorphic processors, Reconfigurable hardware.

I. INTRODUCTION

MOST scientific applications extensively use computationally demanding subroutines similar to the Basic Linear Algebra Subprograms (BLAS) [2] to solve numerical problems. One of the most important BLAS subroutines is General Matrix Multiply (GEMM)[3], defined as: $\mathbf{C} \leftarrow \alpha\mathbf{AB} + \beta\mathbf{C}$, where \mathbf{A} , \mathbf{B} and \mathbf{C} are matrices of dimensions $m \times k$, $k \times n$ and $m \times n$, respectively; α and β are scaling factors. Hereafter, without loss of generality, we shall ignore these scaling factors. Apparently, a faster implementation of the GEMM routine can improve the overall performance of the BLAS and of the applications using it.

In this paper, we address a reconfigurable coprocessor extension of a General Purpose Processor (GPP) for accelerating general matrix multiplication. More specifically, considering the Molen polymorphic architectural paradigm [4], [5], we implement a matrix multiplier as a custom computing unit (CCU) and evaluate its performance on a real hardware prototype. Our design employs the multiplier, proposed in [1], with the major difference that we consider a tightly coupled co-processor architectural paradigm, contrary to the message passing paradigm proposed in [1]. The main contributions of this paper, are:

G. Kuzmanov and W. M. van Oijen are with the Computer Engineering Lab, EEMCS, TU Delft, NL, <http://ce.et.tudelft.nl>, Email: {g.k.kuzmanov,w.m.vanoijen}@tudelft.nl

This work was partially supported by the Dutch Technology Foundation STW, applied science division of NWO (project DCS.7533); and by the European Commission in the context of the Scalable computer ARChitectures (SARC) integrated project #27648 (FP6).

¹ The authors acknowledge Dr. Yong Dou for providing the design files of [1], used as a starting point for this work.

- Complete, reconfigurable, and easily programmable solution of the matrix multiplication problem, based on the Molen polymorphic architectural paradigm.
- A controller that handles all memory access and controls the PEs of the multiplier is proposed.
- Memory access analysis of the algorithm is presented. We formulated a number of equations that should be satisfied in order to achieve peak performance.
- A scalable, flexible, and run-time reconfigurable hardware organization, working with any number of PEs.
- Performance is analyzed as a function of the problem size and supported by real hardware experiments.

We implemented the Molen polymorphic processor running at 300 MHz, with a matrix multiplier with 9 processing elements, organized in a CCU running at 100 MHz on an XC2VP30–6 FPGA. Our experimental results, including all software and hardware overheads, suggest:

- Up to 1.79 GFLOPS sustained system performance, which is 99.2% of the theoretical peak performance.
- Scalable and extendable design over multiple FPGAs.
- Our sustained performance is higher than the performance of related works. The sustained performance converges to the peak performance for large problems.
- Considering a non-optimized BLAS, our design outperforms modern processors, such as Pentium 4 or Athlon 64, by up to 36 times for large problem sizes.

The remainder of this paper is organized as follows. Section II provides implementation details on our proposal. In Section III, we evaluate the design theoretically and by analysis of the results from real hardware experiments. Section IV provides a comparison with related works. Finally, conclusions are presented in Section V.

II. DESIGN DESCRIPTION

To solve $\mathbf{C} \leftarrow \mathbf{AB} + \mathbf{C}$, we employed the block matrix multiplication algorithm [1], which can be efficiently implemented on a linear array of processing elements (PE). The result matrix \mathbf{C} is computed in blocks of $S_i \times S_j$ words, denoted as \mathbf{C}' . Each block \mathbf{C}' is the product of S_i rows of matrix \mathbf{A} and S_j columns of matrix \mathbf{B} , denoted as submatrices \mathbf{A}' and \mathbf{B}' , respectively. The data from matrix \mathbf{A}' are loaded in column-major, the data from \mathbf{B}' - in row-major order in the PEs. Each element of \mathbf{A}' or \mathbf{C}' is transferred to/from one PE only. The data from matrix \mathbf{B} are sent to all PEs, in order to compute S_i products in parallel.

CCU Interface: To allow the integration of the GEMM design into the Molen prototype [6], the former has to adhere to the CCU interface definition [6]. The interface overhead mainly consists of registers to store the parameters, depicted in Fig. 1. The CCU control logic loads all

parameters in the internal registers, then starts the GEMM core unit. When the GEMM core has completed the operation, the `end` signal of the CCU is asserted by the control logic. The core itself consists of a GEMM controller and a number of processing elements.

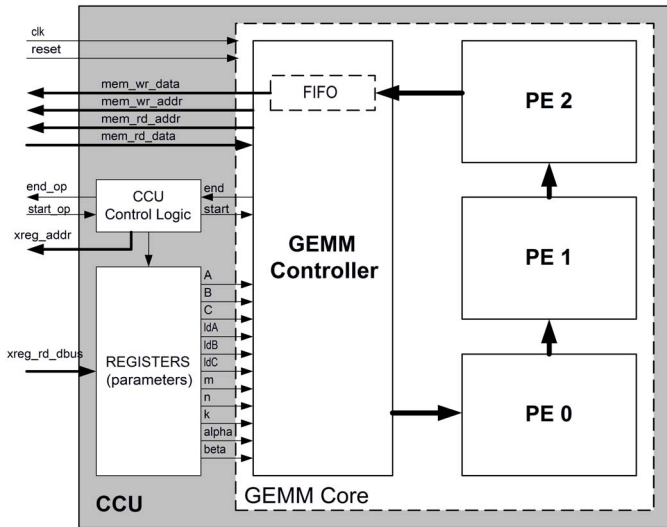


Fig. 1. Structural view of the GEMM core with the CCU interface.

Processing Elements (PE): The GEMM core PEs are the basic building blocks of the matrix multiplier. Each PE contains a floating-point multiply-add unit and two buffers. A memory switching scheme is implemented in the buffers to allow overlapped communication and computation. Multiple PEs can be put together in a linear array, in order to perform more floating-point operations concurrently. More details on the PE organization can be found in [1]. When running in a polymorphic environment, the number of PEs in the CCU can be configured dynamically via the `set` instruction [5]. This capability, together with the scalability of the design, allows more efficient utilization of the available reconfigurable resources

Floating-Point Multiply-Add Units: We consider the double-precision (64-bit) IEEE-754 floating-point format. In our PEs, we used the pipelined MAC unit from [1], with the following modifications:

- Support for the number zero (an unnormalized number) was added.
- The leading-zero counter has been improved and can now execute in one clock cycle. Consequently, the pipeline latency is reduced from 12 to 11 clock cycles.
- Some rounding problems were solved or improved. For example, in the final rounding stage, the exponent is adjusted correctly if overflow of the significand occurs.

Control logic: The DGEMM controller is a key element of the matrix multiplier. It realizes the interface between the PEs, the Molen infrastructure, and the memory subsystem. The tasks of the controller are:

- Divide the input matrices of arbitrary dimensions into sub matrices that can be processed by the PEs.
- Provide efficient read and write access of the PEs to the shared memory.

- Control the linear array of PEs by providing addresses to the local buffers with the correct timing.
- Implement a memory switching scheme such that communication can be overlapped with computations.
- Assert the `end_op` signal when the operation is completed.

We developed a controller that performs all above mentioned tasks, its organization is illustrated in Fig. 2. To save multiplier resources and to increase the clock frequency, all address calculations employ additions only.

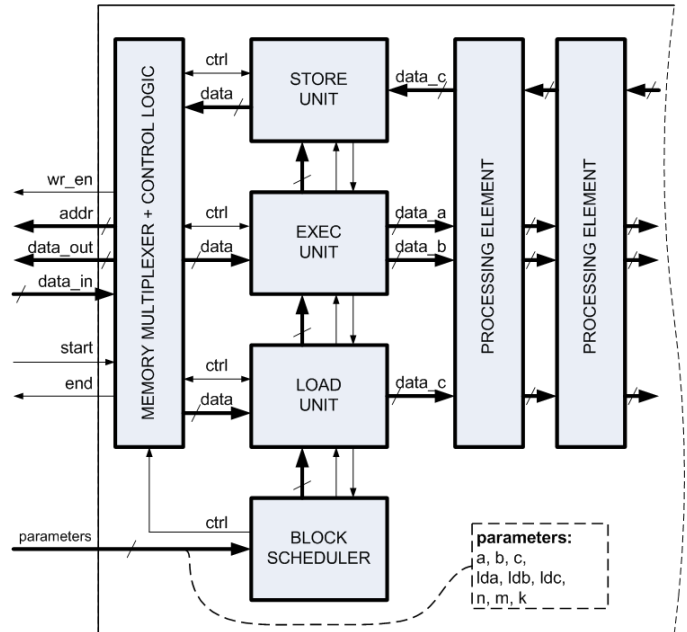


Fig. 2. Structural view of the GEMM Controller.

The **block scheduler** divides the matrices in blocks of $S_i \times S_j$ words and passes start addresses and dimensions of the sub-matrices to the other units. The **load unit** loads the data of sub matrix **C'** from the system memory to the local memories of the processing elements. The **exec unit** loads a column of sub matrix **A'** to the buffers of the PEs, then it loads a row of sub matrix **B'**. Each datum of matrix **B'** is multiplied by the data in the PE buffers. The next column of **A'** is loaded to the second set of buffers, while the PEs are still busy processing the data from the first set of buffers. The active buffers sets are switched continuously to make sure that memory transfers overlap with processing, which improves performance. This loop is repeated k times, k being the number of columns in **A** and the number of rows in **B**. The **store unit** stores the results of sub matrix **C'** back into the system memory, which is shared with the Molen GPP. A **memory multiplexer** controls the system memory accesses as only one unit can read or write at the same time. A state machine keeps track of the status of all units. When the last block has been processed, the controller asserts the `end_op` signal.

GEMM parameters: In software, the Molen programming paradigm constitutes the matrix multiplication as a normal function call. Our function prototype closely resembles the double-precision general matrix multiply

(dgemm) routine of the BLAS - it has the following 11 parameters: **a**, **b** and **c** are the start addresses of the matrices; **m**, **n** and **k** are the dimensions of the matrices; **alpha** and **beta** are scaling vectors. Our hardware implementation currently supports $\alpha \in \{-1, +1\}$ and $\beta \in \{0, 1\}$ only.

When a function is called in the Molen environment, instead of executing its software code, the processor will execute a code using the specific Molen instructions. This code copies the function parameters to the exchange registers and then executes the functionally equivalent hardware operation on the CCU.

III. DESIGN EVALUATION

We first calculate the theoretical upper bound for the performance, based on the external memory bandwidth and then provide experimental results.

Theoretical performance limits: The amount of communication for one $S_i \times S_j$ block is $2S_iS_j + k(S_i + S_j)$ words: $k \times S_i$ words from **A**, $k \times S_j$ - from **B**, and $2 \times S_i \times S_j$ - for matrix **C**. The maximum performance is:

$$P_{\max} = \frac{2BkS_iS_j}{2S_iS_j + k(S_i + S_j)} \quad (1)$$

where B is the bandwidth in [words/s]. As proven in [1], for optimal block sizes $S_i \approx S_j \approx S \Rightarrow$

$$P_{\max} = \frac{BkS}{k + S} \leq BS \quad (2)$$

In practice, however, increasing S will increase P_{\max} if and only if the number of PEs is increased accordingly to match the available bandwidth and storage size.

Real Performance with Memory Switching: At 100 MHz and 64 bits/cyc, the bandwidth equals 800 MB/s. We consider the memory switching scheme, proposed in [1] and analyze the conditions for efficient memory performance. Let $S_i = iP$ and $S_j = S$. So, the parameters i and S describe respectively the number of rows of **A** and columns of **B** that are processed *per PE*. The inner loop of the block matrix multiplication multiplies columns of S_i elements of matrix **A** with rows of S_j elements of matrix **B**. This is repeated k times. In order to initialize the next column of **A** during the computations, so that the PEs do not have to wait between consecutive iterations, the computation time should be greater than or equal to the communication time, i.e.:

$$iS \geq iP + S \quad (3)$$

If $iS > iP + S$, each clock cycle that is not used to load data for the current block can be used to load data for the next block or to store results of the previous block. Thus, the PEs can continue with the next block without stalling, given the following additional requirement:

$$k(iS - (iP + S)) = kiS - kiP - kS \geq 2iPS \quad (4)$$

If (3) and (4) are both satisfied, it guarantees that the data can be processed without stalling. In this case, the

total execution time consists of the following phases: a) Load the first block before computations start; b) Compute all blocks, communication is overlapped with computations; c) Store the results of the last block after the computations have been completed. Consequently, the total execution time can be calculated as:

$$T = \frac{mkn}{P} + 2iPS + iP \quad (5)$$

Note, that (5) holds only when (3) and (4) are both met. Consequently, for optimal performance, the buffer size should be *the lowest possible one, which meets (3) and (4)*. A larger buffer size would reduce the sustained performance because of the increased latency as (5) suggests.

Experimental Results: We have implemented a Molen prototype design with 9 PEs in a XC2VP30-6 FPGA specifying 10ns (100 MHz) as a timing constraint. Table I contains the number of PEs, the parameters S_i

TABLE I
CCU IMPLEMENTATION ON XC2VP30-6(POST PLACE-AND-ROUTE)

PEs	S_i	S_j	Slices	Frequency	$P_{peak}@100\text{MHz}$
1	96	64	2844	101.092 MHz	200MFLOPS
2	96	64	4313	100.301 MHz	400MFLOPS
3	96	64	5726	100.321 MHz	600MFLOPS
4	64	64	7317	100.251 MHz	800MFLOPS
5	80	64	8964	100.271 MHz	1000MFLOPS
6	96	64	10688	100.010 MHz	1200MFLOPS
7	112	64	11843	100.241 MHz	1400MFLOPS
8	64	64	12296	100.251 MHz	1600MFLOPS
9	72	64	13429	100.050 MHz	1800MFLOPS

and S_j , the resource usage (slices) and the frequency after place-and-route. Apparently, 100 MHz was achieved for all considered configurations, resulting in a peak theoretical performance of 1.8 GFLOPS for 9 PEs.

We measured the sustained performance for square matrix multiplications of different dimensions on the Molen prototype. The results are reported in Table II and include the calling overhead in software, the synchronization between the GPP and CCU, and the parameters transfer to/from the exchange registers. Clearly, the sustained performance approaches peak performance for large problem sizes and depends on the number of PEs. For example, with one PE, 95% of the peak performance is sustained for $n = 41$, with 9 PEs - for $n = 142$.

IV. RELATED WORK

A number of matrix multiplication designs for FPGAs were proposed in the past. In [7], a matrix multiplier on an XC2VP125 device with peak performance of 8.3 GFLOPS and external bandwidth of 4.1 GB/s was proposed. Its deep pipeline only handles square matrices of a limited, fixed size. The paper suggests large software overhead and indications of unsolved data hazards. A somewhat improved version of [7] was implemented on a Cray XD1 machine with an XC2VP50 FPGA incorporating 8 PEs running at peak performance of 2.1 GFLOPS [8]. We estimate a potential peak performance of 5.0 GFLOPS for the [8] on a XC2VP125 device. The authors of [9] proposed an

TABLE II
SQUARE MATRIX MULTIPLICATION PERFORMANCE IN MFLOPS. THE CCU RUNS AT 100 MHZ.

n	1 PE	2 PEs	3 PEs	4 PEs	5 PEs	6 PEs	7 PEs	8 PEs	9 PEs
10	143.1	222.7	250.6	286.5	332.2	332.2	332.2	332.2	330.0
20	177.7	319.6	419.9	531.6	613.0	613.0	722.7	722.7	722.7
30	186.1	348.0	490.1	585.8	727.6	828.0	828.0	960.5	959.8
40	189.8	363.3	495.5	658.8	788.7	874.9	982.0	1119.5	1119.5
50	192.0	369.1	523.7	662.5	826.7	901.2	990.5	1099.3	1235.0
60	193.3	374.2	543.8	703.0	852.9	994.2	1084.0	1191.7	1323.0
70	197.0	388.1	558.4	717.6	883.1	998.2	1147.9	1240.9	1350.3
80	197.4	389.7	570.3	759.9	938.0	1062.6	1225.4	1447.0	1562.1
90	197.7	390.9	579.8	748.5	961.8	1121.7	1281.4	1379.6	1629.3
100	198.7	394.7	576.8	765.3	969.6	1102.9	1246.8	1424.4	1533.7
300	199.9	399.7	599.4	799.0	998.2	1197.7	1390.9	1573.4	1758.0
500	200.0	399.9	598.6	799.7	999.6	1189.8	1387.8	1585.4	1783.5
1000	200.0	400.0	598.8	799.9	999.9	1197.4	1398.3	1599.6	1785.2
P_{peak}	200.0	400.0	600.0	800.0	1000.0	1200.0	1400.0	1600.0	1800.0

FPGA-based Hierarchical SIMD (H-SIMD) machine multiplying square matrices only. They employed a memory switching scheme to overlap computations with communications and estimated that 26 PEs would deliver peak performance of 9.36 GFLOPS on XC2VP125. A complete implementation on real hardware is presented in [10] suggesting, in the most optimistic case, a peak performance of 3.4 GFLOPS on XC2VP125. Table III summarizes some characteristics of the designs considered above. In our pro-

TABLE III

COMPARISON WITH RELATED WORK FOR AN XC2VP125 DEVICE.

	[7]	[8]	[9]	[10]	[1]	our
# of PEs	24	25	26	17	39	42
Freq., MHz	172	100	180	100	200	120
GFLOPS	8.3	5.0	9.36	3.4	15.6	10.08
MAC stages	33	19	24	10	12	11
Adder stages	21	19	12	5	8	7
Arbitrary dim.	No	No	No	Yes	No	Yes
Programmability	Hard	Hard	Mod.	Mod.	Hard	Easy

posal, we use the PEs of [1] with some design improvements. The figures in Table III suggest that our design proposal preserves the dominating performance of [1] over related art, both in terms of performance and hardware utilization efficiency. The main architectural difference is that we consider a tightly coupled polymorphic organization with hardwired, fixed control, while [1] assumes a message passing paradigm. Moreover, only simulations and theoretical estimations were presented in [1], while here we provide actual results from real hardware implementations and experiments. In addition, among all considered references, only our design natively handles arbitrary matrix dimensions. Further analysis suggests that our proposal is the easiest to program due to the clean hardware/software Molen polymorphic interface, see Table III.

We also compared our design against three GPP systems: AMD Athlon 64 X2 3800+, 2 GHz, 64 kB L1 data cache, 512 kB L2 cache and 1 GB DDR; Intel Pentium 4, 2.8 GHz, 8 kB L1 data cache, 512 kB L2 cache and 1 GB DDR; and VIA C3, 1 GHz, 64 kB L1 data cache, 64 kB L2 cache and 256 MB DDR. All these systems indicated performance degradation for large matrix sizes, contrary to our design which scales better for large problems. This is a consequence of the limitations of the GPP cache systems. Using highly optimized BLAS (such as ATLAS [11] GotoBLAS [12]), tuned to match the cache sizes, a 3.0 GHz

Pentium 4 could sustain a performance of 5.0 GFLOPS. However, as Table III suggests, our design outperforms even such a highly optimized routines by a factor of 2.

V. CONCLUSIONS

We proposed a holistic design solution of the general matrix multiplication problem. The matrix multiplication design was incorporated in a custom computing unit of the Molen polymorphic processor, and it was implemented on real hardware. We proposed a self-contained polymorphic interface controller that handles all memory accesses and controls the PEs. Furthermore, we analyzed the effect of the design parameters and the problem size on the sustained performance. The conditions to achieve peak performance were expressed in mathematical equations. Experimental results prove that our design is able to achieve near-peak performance for reasonably large problems outperforming related works, including state-of-the-art GPPs.

REFERENCES

- [1] Yong Dou, S. Vassiliadis, G.K. Kuzmanov, and G.N. Gaydadjiev, "64-bit floating-point FPGA matrix multiplication," *Proc. 2005 ACM/SIGDA 13th Int. Symp. on FPGA*, 2005, pp. 86–95.
- [2] J. J. Dongarra, Jeremy Du Croz, Sven Hammarling, and I. S. Duff, "A set of level 3 basic linear algebra subprograms," *ACM Trans. Math. Softw.*, vol. 16, no. 1, pp. 1–17, 1990.
- [3] Bo Kagstrom, Per Ling, and Charles van Loan, "Gemm-based level 3 blas: high-performance model implementations and performance evaluation benchmark," *ACM Trans. Math. Softw.*, vol. 24, no. 3, pp. 268–302, 1998.
- [4] S. Vassiliadis, S. Wong, and S.D. Cofofana, "The Molen μ -coded processor," *Proc. 11th Int. Conf. FPL, LNCS Vol. 2147*, Aug. 2001, pp. 275–285.
- [5] S. Vassiliadis, S. Wong, G.N. Gaydadjiev, K.L.M. Bertels, G. Kuzmanov, and E. Moscu Panainte, "The Molen polymorphic processor," *IEEE Trans. Computers*, pp. 1363–1375, Nov. 2004. <http://ce.et.tudelft.nl/MOLEN/Prototype/index.html>.
- [6] L. Zhuo and V.K. Prasanna, "Scalable and modular algorithms for floating-point matrix multiplication on FPGAs," *Proc. IPDPS 2004*, vol. 1, pp. 92a, 2004.
- [7] L. Zhuo and V.K. Prasanna, "Scalable and modular algorithms for floating-point matrix multiplication on reconfigurable computing systems," *IEEE Trans. Parallel and Distributed Systems*, vol. 18, no. 4, pp. 433–448, 2007.
- [8] Xizhen Xu and Sotirios G. Ziavras, "H-SIMD machine: Configurable parallel computing for matrix multiplication," *Proc. ICCD '05*, pp. 671–676, 2005.
- [9] J. Kadlec and R. Gook, "Floating-point controller as a picoblaze network on a single spartan 3 FPGA," *Proc. MAPLD 2005*, Sept. 2005, pp. 1–11, NASA Office of Logic Design.
- [10] R. C. Whaley and J. Dongarra, "Automatically Tuned Linear Algebra Software," Tech. Rep. UT-CS-97-366, University of Tennessee, Dec. 1997. URL : <http://www.netlib.org/lapack/lawns/lawn131.ps>.
- [11] K. Goto, <http://www.tacc.utexas.edu/resources/software/#blas>.