Matrix Multiplication Implementation in the MOLEN Polymorphic Processor

Wouter M. van Oijen Georgi K. Kuzmanov
Computer Engineering, EEMCS, TU Delft, The Netherlands, http://ce.et.tudelft.nl
Email: {w.m.vanoijen, g.k.kuzmanov}@tudelft.nl

Abstract— **Floating-point matrix multiplication is arguably the most important kernel routine in many scientific applications. Therefore, its efficient implementation is crucial for the overall performance of any computer system targeting scientific computations. In this paper, we propose a holistic solution to accelerate matrix multiplication on reconfigurable hardware using the MOLEN polymorphic processor. The MOLEN polymorphic processor consists of a general purpose processor (GPP) tightly coupled with a reconfigurable coprocessor. The latter can be used to implement arbitrary functions in hardware using custom computing units (CCUs). We implemented matrix multiplication as a CCU for the MOLEN processor and realized it on real reconfigurable hardware. The software interface is defined by the MOLEN programming paradigm, which enables trivial integration of the hardware accelerator at the application level. A matrix multiplication is initiated on the CCU by a MOLEN *execute* instruction and the required operation parameters are transferred through exchange registers. For our experiments, we employ Xilinx Virtex-II Pro technology. An XC2VP30 device proved to be large enough to contain the MOLEN processor infrastructure and a CCU consisting of 9 processing elements, running at 100 MHz. A benchmark application on this system closely approaches the theoretically maximum attainable performance of 1.8 GFLOPS/s. Furthermore, we analyzed the performance with different design parameters and problem sizes. The proposal is clearly scalable and due to its polymorphic nature, it allows optimal configurations in different application contexts and for various chip sizes.**

Keywords— **Field Programmable Gate Arrays, Floating point arithmetic, Matrix multiplication, Reconfigurable architectures.**

I. Introduction

Floating point matrix multiplication is an important kernel in many scientific applications. For example, one very popular scientific problem, which extensively relies on matrix multiplications is solving systems of linear equations. In this work, we consider a version of the Basic Linear Algebra Subprograms (BLAS) [1], where most of the algorithms are rewritten to employ predominantly the GEneral Matrix Multiply (GEMM) routine [2]. Apparently, a faster implementation of this routine can improve the overall performance of the BLAS and thus it could potentially speedup many scientific applications that make use of it. In this paper, we address a reconfigurable coprocessor extension of a general purpose processor (GPP) to accelerate general matrix multiplication. A common shortcoming of many existing reconfigurable hardware proposals, however, is the complex programming interface. To overcome this shortcoming, we consider the MOLEN programming paradigm, which provides a clear hardware/software interface and enables software applications to utilize the hardware matrix multiplication accelerator in a simple and efficient way. Therefore, we have implemented our matrix multiplier as a custom computing unit (CCU) for the MOLEN processor, and tested the performance on a real hardware prototype. Our design is based on the multiplier, proposed in [3] with the main difference that we consider a tightly coupled organization, contrary to [3], where a message passing approach was adopted. The main contributions of this paper are the implementation and experimentation details provided in addition to the main proposal of [4]. We implemented the MOLEN polymorphic processor running at 300 MHz, with the matrix multiplier custom computing unit running at 100 MHz on an XC2VP30–6 FPGA. For this chip, the CCU could incorporate 9 processing elements. We provide experimental results on the MOLEN processor prototype, and measured the sustained matrix multiplication performance (including all software and hardware overheads, such as communication, synchronization, calling overhead, etc.). Our experimental results suggest that:

- The sustained performance for this system is up to 1.79 GFLOPS, which is 99.2% of the theoretical peak performance, calculated to be 1.8 GFLOPS.
- When compared to non-optimized code, e.g. the reference version of the BLAS, our design outperforms modern processors such as Pentium 4 or Athlon 64 by up to 36X for large problem sizes.

The remainder of this paper is organized as fol-

lows. In section 2, we provide background information on the block matrix multiplication algorithm and the MOLEN polymorphic processor. Section 3 provides implementation details on our design. In section 4, we discuss our experimental results from the real hardware implementation. Section 5 provides a comparison with related works. Finally, our conclusions are presented in section 6.

II. Background

A. Block Matrix Multiplication Algorithm

The GEMM routine of the BLAS implements the matrix operation

$$\mathbf{C} \leftarrow \alpha\mathbf{AB} + \beta\mathbf{C} \tag{1}$$

where **A**, **B** and **C** are matrices of dimensions $m \times k$, $k \times n$ and $m \times n$, respectively. The variables α and β are scaling factors. In this paper, without loss of generality, we consider the operation $\mathbf{C} \leftarrow \mathbf{AB} + \mathbf{C}$, i.e. ignoring the scaling factors.

In order to perform a matrix multiplication on the FPGA, data should be exchanged between the main external memory and the on-chip memory. Because of the limited amount of on-chip memory resources, large matrices can not be loaded completely into the device. To deal with this problem, a block matrix multiplication algorithm is employed [3], and it is briefly explained next.

The result matrix C is computed in blocks of $S_i \times S_j$ words. Each block is the product of S_i rows of matrix **A** and S_j columns of matrix **B**, as depicted in Fig. 1. Blocks at the boundary may be smaller. The algorithm in Fig. 2 is used to compute a block **C'**, where **A'**, **B'** and **C'** are sub matrices of **A**, **B** and **C**, respectively (the shaded parts in Fig. 1). This algorithm can be efficiently implemented on a linear array of processing elements, as proposed in [3]. Each value of **A** or **C** is transferred to or from one PE only. Data from matrix **B** is sent to all PEs, in order to compute S_i products in parallel using P processing elements. The data from matrix **A** is loaded in column-major, the data from **B** in row-major order. To improve data reusability, each sub column of **A** is stored in a small buffer. The sub rows of **B** are processed per element so they don't need to be stored in a buffer. Therefore, the total on-chip memory requirement is $S_i \times S_j + S_i$ words. Each PE has a buffer of S_i/P words to store values from **A**, and a buffer of S_iS_j/P words to store values from **C**. The total communication cost for this algorithm (assuming square matrices of $N \times N$) is $\left(\left\lceil \frac{N}{S_i} \right\rceil + \left\lceil \frac{N}{S_j} \right\rceil + 2\right) \times N^2$ words. For more details on the block-based matrix multiplication algorithm, the interested reader is referred to [3].

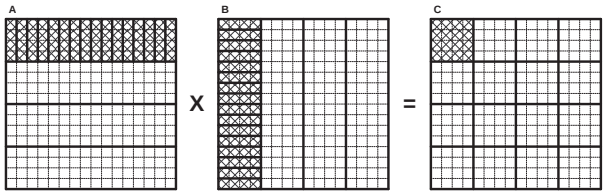

Fig. 1. Example of the block matrix multiplication algorithm, with $m = n = k = 16$ and $S_i = S_j = 4$.

```
for i in 0 to Si-1
   for j in 0 to Sj-1
      load C'(i,j)
for l in 0 to k-1
   for i in 0 to Si-1
      load A'(i,k)
   for j in 0 to Sj-1
      load B'(k,j)
      for i in 0 to Si-1
        C'(i,j) := C'(i,j) + A'(i,l) * B'(l,j)
for i in 0 to Si-1
   for j in 0 to Sj-1
      store C'(i,j)
```

Fig. 2. Pseudo code for the multiplication of one sub block.

B. The MOLEN Polymorphic Processor

Detailed comprehensive explanations of the *MOLEN polymorphic processor* are given in [5], [6]. In this section, we shall only discuss features that are implemented in the MOLEN prototype [7] that we have used and are relevant to our proposal.

The MOLEN polymorphic processor consists of a general purpose processor, also called the *core processor*, and a *reconfigurable coprocessor*. An overview of the organization is depicted in Fig. 3. The reconfigurable processor is used to implement arbitrary functions in hardware in order to speedup applications. A MOLEN reconfigurable operation is split into two phases — *set* and *execute*. `Set` configures the CCU for a particular operation, while `execute`, executes this operation on the configured hardware. This operation is achieved through an extension of the instruction set, called polymorphic ISA. The MOLEN arbiter performs partial decoding of the instructions. The standard instructions are issued to the general purpose processor. The polymorphic `set` and `execute` instructions are redirected to the reconfigurable coprocessor. When an `execute` instruction is issued,

the CCU function is performed on the reconfigurable processor.

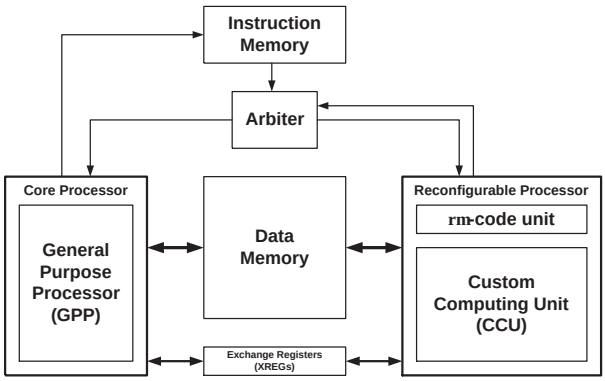

Fig. 3. Structural overview of the MOLEN polymorphic processor.

Communication between the GPP and CCU is done using 32-bit Exchange Registers (XREGs) or the shared data memory. The exchange registers are used to pass function parameters and contain returned values. Larger amounts of data should be stored in and processed from the shared data memory.

III. Design Description

A. Custom Computing Unit (CCU) Microarchitecture

To allow the integration of a design into the MOLEN prototype [7], it has to adhere to the CCU interface defined by the MOLEN architecture. For our experiments, the CCU is implemented as a wrapper for the general matrix multiplication core. The interface overhead mainly consists of registers to store the parameters, depicted in Fig. 4. 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 GEMM core itself consists of a GEMM controller and a number of processing elements. More implementation details on the GEMM corecan be found in [4].

B. Processing Elements

The GEMM processing elements are the basic building blocks of the matrix multiplier. Each PE contains a floating-point multiply-add unit and two buffers (see Figure 5). A memory switching scheme is implemented in the buffers to allow overlapped communication and computation. Multiple processing elements can be put together in a linear array, in order to perform more floating-point operations concurrently.

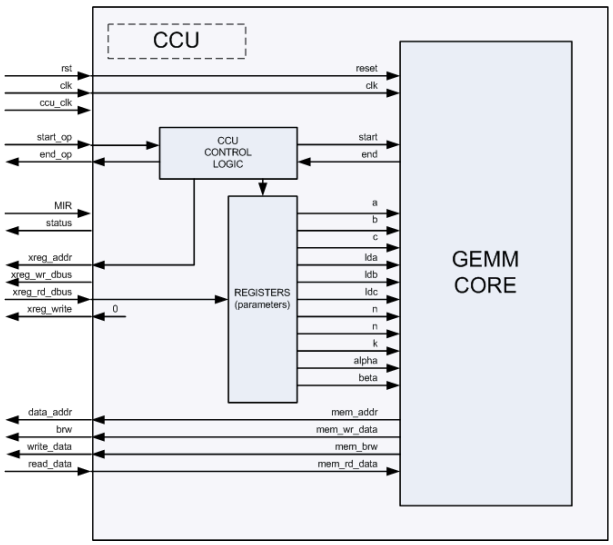

Fig. 4. Structural overview of the CCU organization.

Further details on the PE organization can be found in [3].

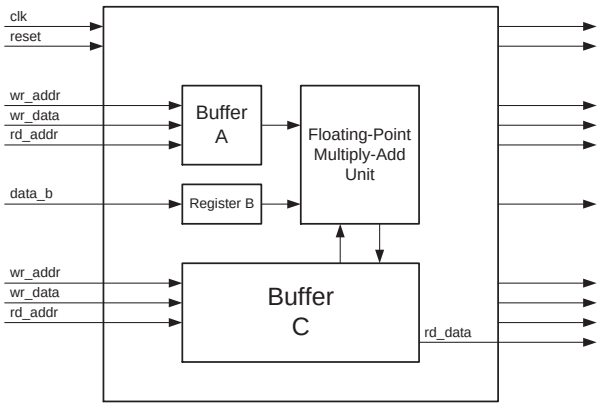

Fig. 5. Structural overview of a Processing Element.

All processing elements are controlled by the Host Controller, although only the first PE receives its control signals directly from the controller. The other elements get the signal from the preceding PE, with one clock cycle delay. This allows the signals to be propagated through the processing element without reducing the maximum frequency. The advantage of this linear array structure is that routing resources scale linearly with the number of processing elements, and that the length of the wires can be kept short enough to achieve high frequencies.

When running in a polymorphic environment, the number of PEs in the CCU can be configured dynamically via the `set` instruction. This capability, together with the scalability of the design, allows more efficient utilization of the available reconfigurable resources and their sharing with other accelerators.

C. Floating-Point Multiply-Add Units

The core of the matrix multiplier is the multiply-add unit. This module performs the operation $C \leftarrow A \times B + C$. The operands are represented in the double-precision (64-bit) IEEE-754 floating-point format [8]. Since computations with 64-bit floating-point numbers are quite complex, the data path should be pipelined to allow high clock frequencies. An overview of the data path is depicted in Fig. 6. Note, that

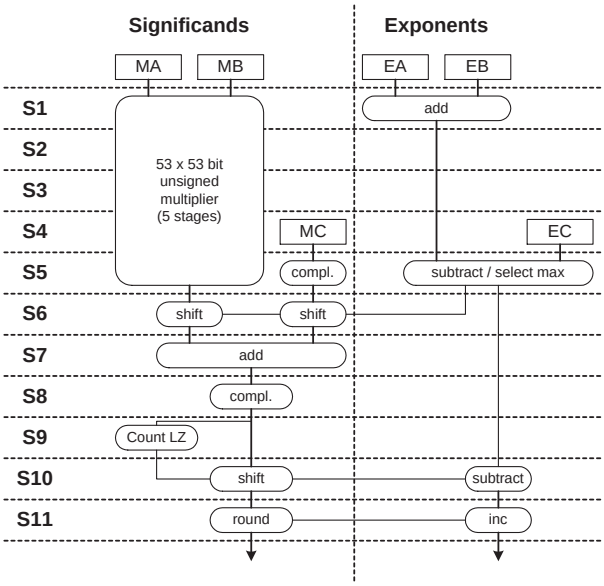

Fig. 6. The pipelined floating-point data path.

the special handling of zero is not shown in the figure. The operands A and B are provided as input signals during the first stage. The third operand is an address for the local memory. This address is used in stage 3 to read operand C. In stage 11, the same address is used to store the result. So, effectively, the product $A \times B$ is added to the contents of a specified address in the local memory.

Although this module is sometimes referred to as "multiply-accumulate" unit, the basic operation it performs is a multiply-add operation. Depending on the controller that supplies the input signals, it could be used for various applications. By repeatedly adding products to the same address, a multiply-accumulate operation can be performed. This is the operation that is implemented as part of the matrix multiplication.

Since there are eight clock cycles between the read and write of that memory location, a data hazard can occur. The host controller prevents this from happening, by supplying new inputs with the correct timing. To improve performance by reducing the number of

```
#pragma call_fpga ccu_dgemm
void ccu_dgemm(long m, long n, long k,
  double alpha, double *a, long lda, double *b,
  long ldb, double beta, double *c, long ldc)
{
  /* actual software implementation here, e.g. */
  int i, j, l;
  double sum;
  for (i = 0; i < m; i++)
  {
    for (j = 0; j < n; j++)
    {
      sum = 0.0;
      for (l = 0; l < k; l++)
        sum += a[lda*i+l] * b[ldb*l+j];
      c[ldc*i+j] = alpha*sum + beta*c[ldc*i+j];
    }
  }
}
```

Fig. 7. Example C source code of a (naive) dgemm implementation.

idle cycles, a reduction scheme such as in [9] could also be employed.

D. Control Logic

The DGEMM controller is a key element of the matrix multiplier. It realizes the interface between the processing elements, the MOLEN infrastructure, and the memory subsystem. The controller organization is originally described in [4]. We just note that it is optimized for high efficiency, not for low bandwidth. Moreover, to save multiplier resources on the FPGA and to increase the clock frequency of the controller, all address calculations are done using additions only.

E. Software Interface

In software, the matrix multiplication can be viewed as a normal function call. Our function prototype closely resembles the `dgemm`[1] routine of the BLAS. The function has 11 parameters, with the following meaning:

- `a`, `b` and `c` are the start addresses of the matrices.
- `m`, `n` and `k` are the matrix dimensions, as defined in section II.
- `alpha` and `beta` are scaling vectors. Currently, our hardware implementation supports only $\alpha \in \{-1, +1\}$ and $\beta \in \{0, 1\}$.

The source code for a possible software implementation is given in Fig. 7. The `#pragma` statement before the function definition tells the compiler that a hardware implementation of this routine is available.

When the function is called in the MOLEN environment, instead of executing the function specified

[1] dgemm = double-precision general matrix multiply

```
xreg0   ← m
xreg1   ← n
xreg2   ← k
xreg3   ← alpha(hi)
xreg4   ← alpha(lo)
xreg5   ← a
xreg6   ← lda
xreg7   ← b
xreg8   ← ldb
xreg9   ← beta(hi)
xreg10  ← beta(lo)
xreg11  ← c
xreg12  ← ldc
sync
fpga_execute[ccu_dgemm]
```

Fig. 8. MOLEN pseudo code for the dgemm routine.

TABLE I
IMPLEMENTATION RESULTS (POST PLACE-AND-ROUTE) FOR THE CCU WITH DIFFERENT NUMBER OF PROCESSING ELEMENTS, AT 100 MHZ.

PEs	S_i	S_j	**Slices**	**Frequency**
1	96	64	2844	101.092 MHz
2	96	64	4313	100.301 MHz
3	96	64	5726	100.321 MHz
4	64	64	7317	100.251 MHz
5	80	64	8964	100.271 MHz
6	96	64	10688	100.010 MHz
7	112	64	11843	100.241 MHz
8	64	64	12296	100.251 MHz
9	72	64	13429	100.050 MHz

by the C source code, the processor will execute functionally equivalent code using MOLEN instructions. This code copies the function parameters to exchange registers and then executes the hardware operation on the CCU. Example pseudo code for the same routine for a MOLEN architecture is listed in Fig. 8.

IV. Experimental Results

We have implemented a test framework including the proposed matrix multiplier design with multiple processing elements in an XC2VP30–6 FPGA. This device contains 13,616 slices and 136 embedded multipliers. Table I contains the number of processing elements, the parameters S_i and S_j, the resource usage (slices) and the frequency after place-and-route. We specified a frequency of 100 MHz in the user constraints file, and the results indicate that this was achieved for all our configurations. We succeeded to fit 9 processing elements in this relatively small FPGA device, resulting in a peak performance of 1.8 GFLOPS at 100 MHz. Synthesis results estimated a maximum possible frequency of 134 MHz for the largest configuration, which would deliver 2.4 GFLOPS on this device. We did not consider harder design optimizations to reach higher frequencies, but rather ran all our configurations at 100 MHz.

By extrapolating the results for a larger device such as XC2VP125, we can safely predict a performance of over 10 GFLOPS on a single FPGA obtained by 42 processing elements, running at 120 MHz.

We used a test application to measure the sustained performance for matrix multiplications of different dimensions on a MOLEN processor prototype. Our measurements include all overheads, such as calling overhead in software, synchronization between GPP and CCU, and transferring the parameters to and from the exchange registers. For square matrix multiplications, the results are plotted in Fig. 9. Clearly, the sustained performance approaches peak performance for large problems. The problem size for which the real sustained performance is close enough to the theoretical peak performance depends on the number of PEs. For example, our measurements indicate that with one processing element, 95% of the peak performance can be sustained for $n = 41$. With 9 processing elements, the same efficiency is achieved for $n = 142$.

V. Related Work

In the past, a number of matrix multiplication designs for Field-Programmable Gate Arrays (FPGAs) were proposed or implemented. Most papers only include simulation results or an evaluation of the theoretical peak performance and do not provide reports of sustained performance on real hardware implementations. In [3], the authors proposed a matrix multiplier design that could potentially achieve up to 15.6 GFLOPS on a Virtex-II Pro XC2VP125 FPGA. However, as pointed out in [10], this numbers do not include any control overhead, and the sustained performance in real hardware was not measured or precisely evaluated. Also, the designers assumed that communication can be overlapped with computations, but they did not mention the conditions for that, or what performance can be potentially achieved under non-optimal conditions. We shall compare our proposal to the most important contributions according to our best knowledge. A summary of the related art we consider is presented below:

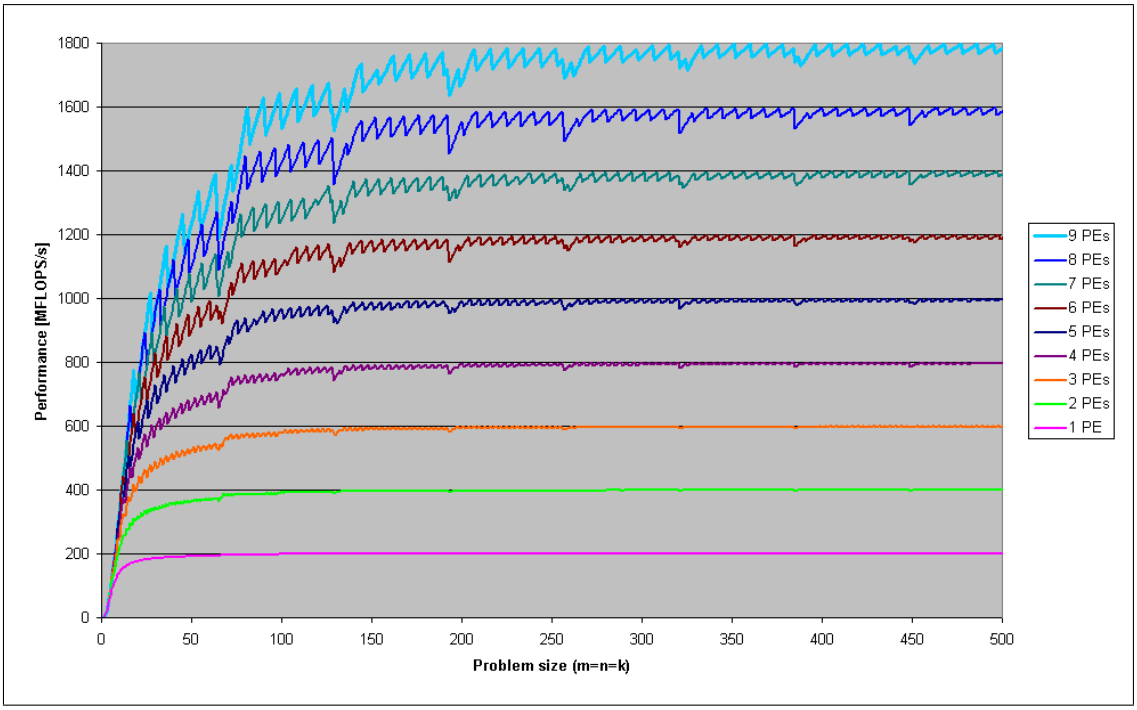

Fig. 9. Square matrix multiplication performance.

• In [11], a matrix multiplier design was proposed with a peak[2] performance of 8.3 GFLOPS on an XC2VP125 device with an external bandwidth of 4.1 GB/s. However, that design only handles square matrices of a limited, fixed size. For smaller matrices the performance is lower because not all processing elements are used. For larger matrices, a block matrix algorithm is required, resulting in large software overhead. It is unclear how non-square matrices are processed. The authors also indicate a data hazard, but do not present a solution for this. Furthermore, their floating-point units have a deep pipeline: 12 stages for the multiplier and 21 stages for the adder. These issues have a negative effect on the overall performance. However, the overall performance, including software overhead, is not discussed in the paper.

• In [12], the same authors present an improved version of their design from [11]. The number of pipeline stages was reduced to 8 stages for the multiplier and 11 stages for the adder. A new algorithm was proposed that could handle matrix dimensions larger than the number of PEs. Their design was implemented on a Cray XD1 machine with an XC2VP50 FPGA. The device could incorporate 8 processing elements running at 130 MHz for a peak performance of 2.1 GFLOPS. We estimate a peak performance of 5.0 GFLOPS for the same design on a XC2VP125 device.

• The authors of [10] proposed an FPGA-based Hierarchical SIMD (H-SIMD) machine. They also employed a memory switching scheme to overlap computations with communications. To test the effectiveness of the design, a matrix multiplication was implemented. Using commercial IP-cores from Quixilica for the floating-point units, they were able to fit 16 processing elements — called nano-processors — in an XC2V6000 FPGA, running at 148 MHz. They estimated that 26 processing elements running at 180 MHz would fit in an XC2VP125 device. This would deliver a peak performance of 9.36 GFLOPS. Only square matrix multiplications are considered, and the matrices should be padded to a multiple of the block size, which lowers the performance slightly.

• A complete implementation on real hardware is presented in [13]. The authors only provide performance results for a 18-bit vector product implementation, but also include synthesis results of the commercial IP cores from Celoxica that are employed. Based on the reported numbers, we estimate that in the most optimistic case a peak performance of 3.4 GFLOPS could be achieved on an XC2VP125 device.

Our proposal adopts and improves the processing elements microarchitecture from [3], which is the fastest design among the presented above. However, we propose a tightly-coupled processor-coprocessor architectural paradigm, which is fundamentally dif-

[2]The authors of [11] refer to peak as to *sustained performance*, but they use different definitions than ours.

ferent from the proposed message passing communication interface, considered in [3]. We also note that our design provides some functional and design improvements of [3]. Last, but not least, our proposal was evaluated on a real hardware prototype, while the design, reported in [3] was evaluated through simulations only without taking into account hidden communication delays and implementation overheads. Therefore, our experimental results suggest realistic performance and implementation figures.

We also compared the performance of our design with the performance of some contemporary general purpose processors. We measured the performance of three different platforms for the same problem sizes, using reference code from the BLAS, translated into C. Admittedly, this code is not very optimized, so it suggests typical performance only without system-dependent optimizations. Fig. 10 depicts a performance comparison of our design with 1 to 9 processing elements, against the following three general purpose systems:

- AMD Athlon 64 X2 3800+ ("Windsor") at 2.0 GHz with 64 kB L1 data cache, 512 KB L2 cache and 1 GB DDR memory
- Intel Pentium 4 ("Northwood") at 2.8 GHz with 8 kB L1 data cache, 512 kB L2 cache and 1 GB DDR memory
- VIA C3 ("Nehemiah") at 1.0 GHz with 64 kB L1 data cache, 64 kB L2 cache and 256 MB DDR memory

All these systems indicated performance degradation for large matrix sizes. This is a consequence of the complex memory hierarchy that modern processors have, more precisely, due to the limitations of their caches. Optimized versions of the BLAS, such as ATLAS [14] or the GotoBLAS [15] use a block matrix algorithm with the block size tuned to match the caches of the processor. Using such libraries, a 3.0 GHz Pentium 4 could sustain performance of 5.0 GFLOPS. However, using larger FPGAs, we can still outperform such a highly optimized routine by a factor of 2. It is also worth noting that the computational capacity of the FPGAs is expected to increase faster than that of general purpose processors [16], [17], which will likely increase the advantage of our proposal against future GPPs.

VI. Conclusions

In this paper, we proposed a design solution of the general matrix multiplication problem based on the MOLEN polymorphic architecture. A previously proposed matrix multiplication design was incorporated in a custom computing unit (CCU) of the MOLEN polymorphic processor, and it was implemented on real hardware. We exploited the tightly coupled processor-coprocessor MOLEN paradigm in order to obtain high performance. Moreover, we improved the design of the processing elements in order to allow their complete functionality at high clock frequencies and low resource usage. A benchmark application on the MOLEN processor was used to measure the real matrix multiplication performance. Experimental results prove that our design is able to achieve near-peak performance for reasonably large problems. On the relatively small XC2VP30 FPGA, running at 100 MHz, we sustained 1.79 GFLOPS. Using a large FPGA, sustained performance of over 10 GFLOPS can be expected, thereby outperforming related art, including highly optimized applications running on state-of-the-art general purpose processors. Thanks to the design scalability and the polymorphic nature of the considered architecture, our proposal allows efficient implementations in various application contexts and optimal utilization of any available reconfigurable resources.

Acknowledgment

We acknowledge Dr. Yong Dou for providing us the design files of [3], which we utilized as starting point for our work. This work was partially supported by the Dutch Technology Foundation STW, applied science division of NWO; by the Technology Program of the Dutch Ministry of Economic Affairs (project DCS.7533); and by the European Commission in the context of the Scalable computer ARChitectures (SARC) integrated project #27648 (FP6).

References

[1] 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.

[2] 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.

[3] Yong Dou, S. Vassiliadis, G.K. Kuzmanov, and G.N. Gaydadjiev, "64-bit floating-point fpga matrix multiplication," in *FPGA '05: Proc. 2005 ACM/SIGDA 13th international symposium on Field-programmable gate arrays*, New York, NY, USA, 2005, pp. 86–95, ACM Press.

[4] G. Kuzmanov and W. M. van Oijen, "Floating-point matrix multiplication in a polymorphic processor," in *Proc. IEEE International Conference on Field-Programmable Technology (ICFPT'07)*, December 2007.

[5] S. Vassiliadis, S. Wong, and S.D. Cotofana, "The

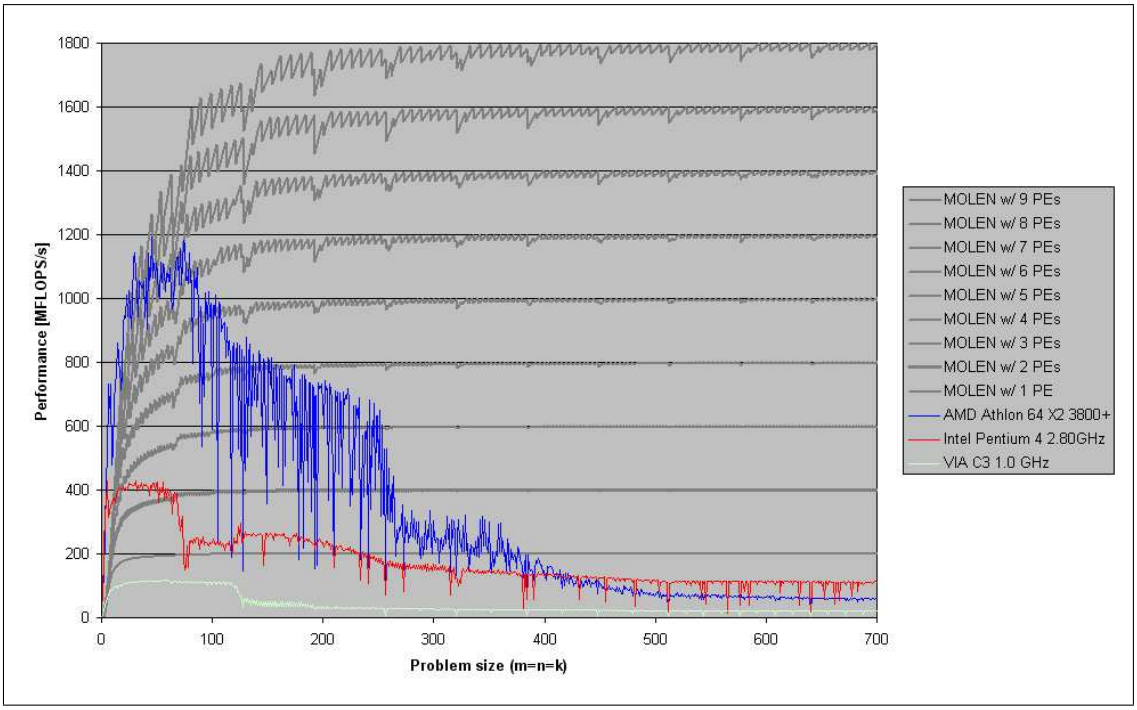

Fig. 10. Square matrix multiplication performance on MOLEN compared to general purpose processors.

molen $\rho\mu$-coded processor," in *in 11th International Conference on Field-Programmable Logic and Applications (FPL), Springer-Verlag Lecture Notes in Computer Science (LNCS) Vol. 2147*, August 2001, pp. 275–285.

[6] S. Vassiliadis, S. Wong, G.N. Gaydadjiev, K.L.M. Bertels, G.K. Kuzmanov, and E. Moscu Panainte, "The molen polymorphic processor," *IEEE Transactions on Computers*, pp. 1363–1375, November 2004.

[7] Delft University of Technology, "The molen prototype," http://ce.et.tudelft.nl/MOLEN/Prototype/index.html.

[8] "IEEE standard for binary floating-point arithmetic.," .

[9] Michael R. Bodnar, John R. Humphrey, Petersen F. Curt, James P. Durbano, and Dennis W. Prather, "Floating-point accumulation circuit for matrix applications," *fccm*, vol. 0, pp. 303–304, 2006.

[10] Xizhen Xu and Sotirios G. Ziavras, "H-simd machine: Configurable parallel computing for matrix multiplication," in *ICCD '05: Proc. 2005 International Conference on Computer Design*, Washington, DC, USA, 2005, pp. 671–676, IEEE Computer Society.

[11] Ling Zhuo and Viktor K. Prasanna, "Scalable and modular algorithms for floating-point matrix multiplication on fpgas," *ipdps*, vol. 01, pp. 92a, 2004.

[12] Ling Zhuo and Viktor K. Prasanna, "Scalable and modular algorithms for floating-point matrix multiplication on reconfigurable computing systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 18, no. 4, pp. 433–448, 2007.

[13] J. Kadlec and R. Gook, "Floating point controller as a picoblaze network on a single spartan 3 FPGA," in *MAPLD 2005 International Conference Proceedings*, R. B. Katz, Ed., Washington, September 2005, pp. 1–11, NASA Office of Logic Design.

[14] R. Clint Whaley and Jack Dongarra, "Automatically Tuned Linear Algebra Software," Tech. Rep. UT-CS-97-366, University of Tennessee, December 1997, URL : `http://www.netlib.org/lapack/lawns/lawn131.ps`.

[15] Kazushige Goto, ," http://www.tacc.utexas.edu/resources/software/#blas.

[16] Keith D. Underwood and K. Scott Hemmert, "Closing the gap: Cpu and fpga trends in sustainable floating-point blas performance," in *FCCM '04: Proc. 12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, Washington, DC, USA, 2004, pp. 219–228, IEEE Computer Society.

[17] Keith Underwood, "Fpgas vs. cpus: trends in peak floating-point performance," in *FPGA '04: Proc. 2004 ACM/SIGDA 12th international symposium on Field programmable gate arrays*, New York, NY, USA, 2004, pp. 171–180, ACM Press.