Strassen's Matrix Multiplication for Customisable Processors

H.M.D. Ip[1], J.D. Low[2], P.Y.K. Cheung[2], G. Constantinides[2], W. Luk[1], S.P. Seng[1] and P. Metzgen[3]

[1]*Department of Computing, Imperial College, 180 Queen's Gate, London, SW7 2BZ, UK*
[2]*Department of EEE, Imperial College, Exhibition Road, London, SW7 2BT, UK*
[3]*Altera European Technology Centre, High Wycombe, Bucks, HP12 4XF, UK*

Abstract

Strassen's algorithm is an efficient method for multiplying large matrices. We explore various ways of mapping Strassen's algorithm into reconfigurable hardware that contains one or more customisable instruction processors. Our approach has been implemented using Nios processors with custom instructions and with custom-designed coprocessors, taking advantage of the additional logic and memory blocks available on a reconfigurable platform.

1. Introduction

Recent advances in reconfigurable hardware technology enable one or more instruction processors to be implemented in a single device [1]. Such reconfigurable implementations support customisation for specific applications in two main ways:

- an instruction set that can be extended with custom instructions, and
- a custom-designed co-processor.

The purpose of this paper is to explore the use of such customisable processor technology in implementing Strassen's algorithm [2], an efficient method for multiplying large matrices. Other algorithms such as K-means clustering [3] have also been targeted for reconfigurable instruction processor implementation.

The remainder of the paper is organised as follows. Section 2 introduces Strassen's algorithm for matrix multiplication. Section 3 describes the custom instruction facility for the Nios processor [5], and its use in implementing Strassen's algorithm. Section 4 explains how a custom-designed co-processor can be developed to support matrix multiplication. Section 5 presents a framework for concurrent processing based on multiple copies of customisable instruction processors, and shows how it benefits Strassen's algorithm. Finally Section 6 summarises our approach, and discusses opportunities for further research.

2. Strassen's Algorithm

Strassen's algorithm [2] and its variants are known to be among the most efficient matrix multiplication methods. It reduces the number of scalar multiplications involved in the computation of a matrix multiplication. This is achieved by replacing a large matrix multiplication with a combination of smaller matrix multiplications and matrix additions. These smaller multiplications can also be subdivided using the algorithm recursively.

Conventional matrix multiplication requires n^3 scalar multiplications for an n by n matrix. Strassen's algorithm requires $4(n-1)^3$ scalar multiplications, together with extra additions. The procedure is summarised as follows. Let A and B be two n by n matrices, where n is an even integer. Partition the two input matrices A and B and the result matrix C into quadrants as follows [6]:

$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \bullet \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$$

The symbol $\bullet$ in the above equation represents matrix multiplication. We then compute the four quadrants of the result matrix as follows:

$$\begin{aligned} C_{11} &= P_1 + P_4 - P_5 + P_7 \\ C_{21} &= P_2 + P_4 \\ C_{12} &= P_3 + P_5 \\ C_{22} &= P_1 + P_3 - P_2 + P_6 \end{aligned}$$

$$\begin{aligned} P_1 &= (A_{11} + A_{22}) \bullet (B_{11} + B_{22}) \\ P_2 &= (A_{21} + A_{22}) \bullet B_{11} \\ P_3 &= A_{11} \bullet (B_{12} - B_{22}) \\ P_4 &= A_{22} \bullet (B_{21} - B_{11}) \\ P_5 &= (A_{11} + A_{12}) \bullet B_{22} \\ P_6 &= (A_{21} - A_{11}) \bullet (B_{11} + B_{12}) \\ P_7 &= (A_{12} - A_{22}) \bullet (B_{21} + B_{22}) \end{aligned}$$

Our tests show that Strassen's algorithm does improve a pure software implementation of a matrix multiplication.

3. Custom Instructions

Reconfigurable systems, by their very nature, are more flexible than fixed-function systems. Configurable instruction processors allow many forms of customisation. For instance, source code running on the processor can be

0-7803-7574-2/02/$17.00 ©2002 IEEE

optimised, special interfaces to custom designs can be included, and features not used by the processor can be removed. However the implementation of such solutions may be time consuming and complex.

The Nios configurable processor [5] provides a simple solution to the problem of incorporating new hardware to a processor and in accessing the hardware from a software program. The Nios instruction set architecture can be extended to include custom instructions, and this has several advantages:

1)Integrated into the instruction processor pipeline;
2)Direct register interaction;
3)States/data can transcend multiple instruction calls;
4)Hardware size limited only by chip size.

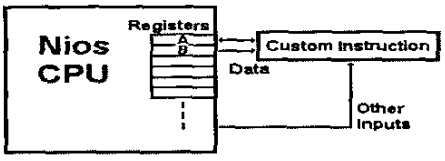

Figure 1 Nios custom instruction architecture.

The Nios custom instruction takes two 32-bit words from registers A and B and outputs one in register A (Figure 1). There are additional inputs that can be used to control other aspects of the custom instruction's functionality. The close integration through the CPU registers may limit data throughput to main memory, but if the required calculation is computationally complex and required at ad hoc intervals, the custom instruction provides a very good opportunity for speeding up software algorithms, as the current data may already be loaded in the registers.

To improve performance of common multiply-accumulate based algorithms, we propose the use of a multiply-add routine in a custom instruction. An internal accumulator is used to reduce memory bandwidth requirements and thus improves the overall performance.

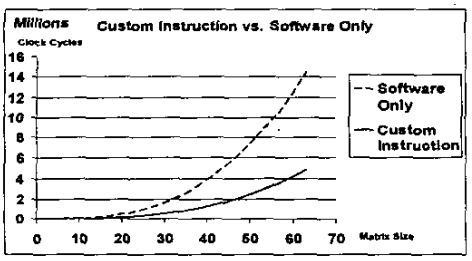

Figure 2 Custom instruction test results.

It is clear from Figure 2 that the custom instruction with the built-in accumulator performs significantly better than a software implementation. Performance increases of up to 77% are observed for large matrices. When we use Strassen's algorithm with the custom instruction, little performance increase is achieved. Strassen's algorithm only becomes consistently faster for matrices of size larger than 64.

4. Matrix Multiplication Co-processor

The deficiency of a general-purpose processor when processing computationally intensive scientific algorithms is frequently compensated by installing an additional specialized processing unit. In our experiment, we demonstrate the performance advantage of an additional hand-optimised co-processor in a system with a customisable processor.

To implement Strassen's algorithm, the co-processor is designed as a stand-alone matrix multiplier and can produce up to one row of the result matrix at a time. The matrix addition part of the algorithm is done by the main processor. The matrix multiplication co-processor is designed to stream matrix data from memory. Its sequence of operations starts with streaming a single row from the first matrix operand into its internal buffers. It next streams the whole second matrix operand, calculating the first row of the result matrix at the same time (Figure 3). Provided that data can be supplied continuously at every cycle, $n(n^2 + 2n + 6)$ cycles are required to calculate the result of two n by n operands, including co-processor setup.

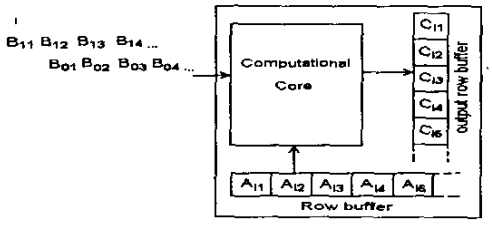

Figure 3 Matrix co-processor architecture.

The matrix co-processor is efficient for a larger stream of data, as the overhead of initiating a DMA is offset by more efficient memory access, as compared to the custom instruction or software implementations. With a buffer size of 8, a performance increase of 45% has been measured. However, with larger buffer sizes, simulations show that performance increases can be up to 86% (Figure 4).

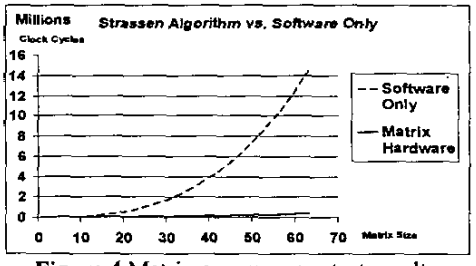

Figure 4 Matrix co-processor test results.

When we use Strassen's algorithm with the matrix co-processor, we find that the performance is lower than using the hardware alone. This is due to two factors. First, the additions are computed by the Nios, which takes a rather long time. Second, the Nios and the matrix co-processor are competing for memory bandwidth. Strassen's algorithm does have the advantage that it allows matrices of twice the size to be computed using the same piece of hardware.

Further performance gains can be obtained with more co-processors, provided that the memory bandwidth is sufficient to keep the processing units busy. Here we propose a memory architecture suitable to be used under a programmable platform to overcome the memory bandwidth bottleneck.

An example of one processing unit is used to illustrate the idea. As shown in Figure 5, we have two additional blocks of memory installed as buffers for the co-processor. The overall mechanism is like a swinging buffer commonly found in video processing architectures. Calculation starts with the general processor evaluating partial matrix sums (equations for P1 to P7), with results stored in buffer1. ($A_{11}+A_{22}$ and $B_{11}+B_{22}$ for P1, say.) The general processor then starts calculating the third partial sum, results stored in buffer2. At the same time, the co-processing unit could start evaluating the first set of matrix multiplication in P1 by streaming from buffer1. In this way, the main processor will not need to compete with the co-processing unit for memory resources. Our tests on Strassen's algorithm show that for processors with load/store architecture, matrix addition requires more cycles than the predicted n^2 cycles. In the worst case, the adder in the general processor cannot keep up with the multiplier in the co-processor. Therefore, the workloads in the two processing units are fairly balanced.

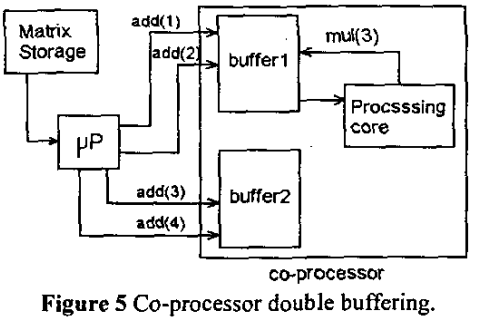

Figure 5 Co-processor double buffering.

5. Multi-Customisable Processors

We show that it is feasible to implement a multi-processor system on an advanced programmable logic platform. Additional logic is available for system bus arbitration circuit and on-chip memory blocks. The customisability of a processor on a programmable chip facilitates the development of a synchronization framework for distributed programming on multiple processors. The system implemented consists of two general customisable processors; however, the principles could easily be extended to cover more processors provided that there are sufficient resources on the programmable chip.

For multiple processors sharing data structures, some synchronization techniques are often required to ensure that no more than one processor is working on the same data structure at any given time. Common techniques include locks, semaphores and monitors [4]. In our implementation, the processors share common registers to keep track of which processor is accessing which shared data block. Software running on the processors in the system has the responsibility of checking these registers, so that it does not run into occupied shared data. This is accomplished by an atomic "test and set" custom instruction operating on the shared registers. In this way, a processor can obtain control of a "free" data structure by writing to its "lock" flag in an uninterruptible way.

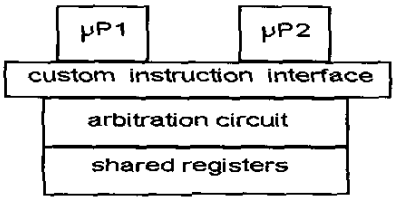

Figure 6 Arbitration through custom instructions.

To demonstrate the use of our hardware-assisted synchronization framework, we recode the first recursion level of Strassen's matrix multiplication algorithm as a distributed program for our dual processor platform. The aim is to balance the computation load across the processors. At some point in the program, the two processors are competing to run the next step of Strassen's algorithm. This is supported by the custom "test and set" instruction. With a limited memory bandwidth, the dual processor achieves a performance gain of almost 25% over a single processor (Figure 7).

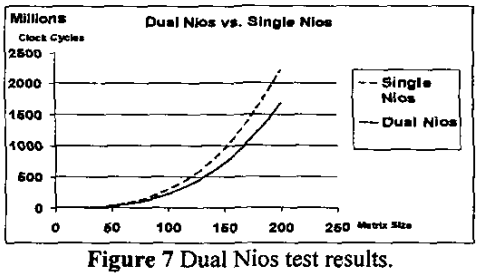

Figure 7 Dual Nios test results.

Further performance gain can be achieved by introducing a new memory architecture that will increase the memory bandwidth of the whole system. After closely

inspecting Strassen's algorithm, we derive a novel memory architecture suitable for parallel processing with dual processors. The first seven equations (P1-P7, Section 2) consist of ten matrix additions and seven matrix multiplications. We employ four memories, (A, B, A', B') using on-chip resources (memory blocks and bus arbitration logic) or off-chip memory bank connections.

The first step is to generate the matrix sums for the multiplication stage. Initially operand matrix A (A_{11}, A_{12}, A_{21}, A_{22}) is stored in memory block A, and operand matrix B (B_{11}, B_{12}, B_{21}, B_{22}) is stored in memory block B. The aim of the operations is that at any instance in the sum and multiplication stages, we have no more than one processor working on the same memory block.

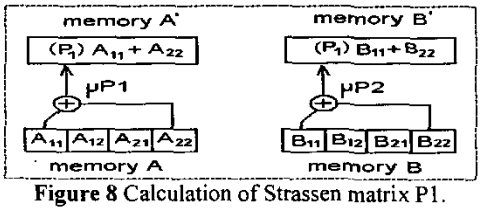

Figure 8 Calculation of Strassen matrix P1.

Figure 8 shows the first set of calculations: the partial sums ($A_{11} + A_{12}$) and ($B_{11} + B_{22}$) are calculated by the two processors and stored in memory blocks A' and B' respectively. Figure 9 and Figure 10 show how the partial sums of P6 and P7 are calculated and stored. The rest of the partial sums are calculated and stored in appropriate memory blocks, such that the multiplication operations in equation (P6,P5), (P2,P3) and (P1,P4) can be executed in parallel without both processors working on the same memory block at the same time.

For example, the partial sums of equation P6 and P5 are stored in memory block B' and memory blocks (A,B) respectively. To perform the matrix multiplication of the partial sums on P6 and P5, one processor would access only memory block B' (P6), while the other processor will be working on blocks A and B (P5) (Figure 11). Hence the two processors would be working on different memory blocks. With results of P1-P6 stored in A, B and B', the final matrix multiplication in P7 can be executed on block A' only. At the same time when the multiplication stage of P7 is evaluated, C_{12}, C_{21} and C_{22} can be calculated; no two processors work on the same block of memory at the same time. A more fine-grained approach is to split the multiplication in P7 and have two processors working on it. However, given the limited memory bandwidth, this approach is unlikely to improve performance by much.

In conclusion, with an increasing number of processors in the system and provided that there is sufficient spare hardware on the programmable platform to implement additional memory blocks and bus arbitration circuit, the performance of distributed software could benefit significantly from an on-chip multiprocessor system.

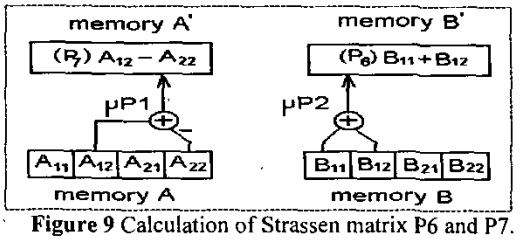

Figure 9 Calculation of Strassen matrix P6 and P7.

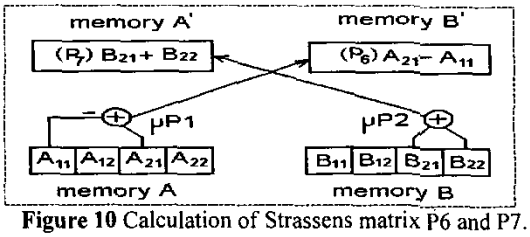

Figure 10 Calculation of Strassens matrix P6 and P7.

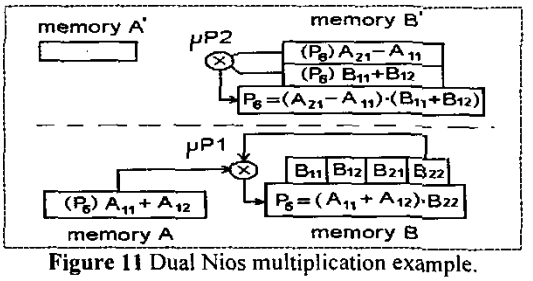

Figure 11 Dual Nios multiplication example.

6. Summary

We have presented various ways of mapping Strassen's algorithm into reconfigurable hardware containing one or more customisable instruction processors. Current and future work includes studying the scalability aspects of our approach, and generalising our techniques to cover other applications.

References

[1] Altera Corp., *Altera Programmable Data Book*, 2001.

[2] V. Strassen, "Gaussian elimination is not optimal", *Numerische Mathematik*, Vol. 13, pp. 354-356, 1969.

[3] M. Gokhale et al, "Early experience with a hybrid processor: K-means clustering", *Proc. Int. Conf. on Engineering of Reconfigurable Systems and Algorithms (ERSA01)*, CSREA Press, 2001.

[4] J. Magee and J. Kramer, *Concurrency: State Models and Java Programs*, Wiley, 1999.

[5] Altera Corp., *Nios Tutorial*, 2002.

[6] M. Thottethodi, S. Chatterjee and A.R. Lebeck, "Tuning Strassen's matrix multiplication for memory efficiency", *Proc. High Performance Networking and Computing Conference (SC98)*, 1998.