MULTITHREADED SYSTOLIC COMPUTATION

Radovan Sernec, Matej Zajc, Jurij Tasič
Faculty of Electrical Engineering, University of Ljubljana
Tržaška 25, SI-1000 Ljubljana, Slovenia
zajcm@fe.uni-lj.si, http://ldos.fe.uni-lj.si/

ABSTRACT

In this paper we propose a synergy of processing on parallel processor arrays (systolic or SIMD) and multithreading, termed multithreaded systolic computation.

The multithreaded systolic computation principle is demonstrated on a programmable systolic array executing a set of linear algebra algorithms. We demonstrate that multithreaded systolic computation can provide throughput improvements that asymptotically approach the number of simultaneously executable threads.

Keywords: multithreading, homogeneous processor array, systolic array, systolic algorithm, linear algebra.

1. INTRODUCTION

DSP, image processing as well as linear algebra algorithms have found an efficient implementation medium in parallel computing domain. These algorithms can be efficiently mapped onto array processors (systolic arrays, wavefront arrays and SIMD arrays), due to their regularity on data level [1, 2]. Systolic arrays were designed to tackle fine-grained computation and exploit data level parallelism of these problems directly. They feature a very homogeneous (regular) design and consist of a number of identical processing elements.

Our paper is focused on systolic arrays with programmable processing elements. The model of a programmable processing element is the central point of the simultaneous, concurrent execution of several algorithms on one systolic array. In order to achieve consistent speedups with programmable and pipelined processing elements we execute multiple algorithms simultaneously on a systolic array, termed multithreaded systolic computation [3].

Multithreading is a technique for hiding various types of latencies and is used in contemporary operating systems, Java programs, etc. It is seen as the technique that can substantially push forward the throughput of 21st century single processors [4]. The term thread refers to a single path of execution or control flow. There are several flavours of multithreading, which differ by thread switching intervals, synchronization mechanisms among threads and implementation requirements. On the finer granularity level of processor clock cycles, multithreading can hide the latencies of jump and branch or even long latency instructions as division or square root. This kind of implementation is covered in our contribution.

2. THROUGHPUT LIMITATIONS OF SYSTOLIC ALGORITHMS

Systolic computation combines parallel processing and pipelining. Parallel processing is the result of parallel operation of all processing elements within a systolic array that co-operate in order to solve the problem faster. Pipelining is done on the level of the processing elements within the systolic array structure.

Each systolic algorithm viewed from the processing element perspective includes three distinct processing phases: data input, algorithm processing, data output. All three phases constitute what is known as a systolic cycle. The systolic cycle is defined by a global clock, which synchronises data exchange between the processing elements within the systolic array.

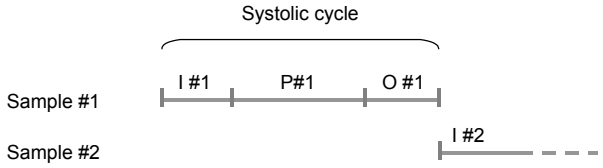

Figure 1: Systolic processing phases within a processing element during systolic cycle. I: input, P: processing, O: output.

Throughput of a systolic array can be limited due to various reasons: low systolic algorithm efficiency, data synchronization between parts of the systolic array with different types of processing elements [1-3], data dependencies and long latency operations within a processing element. The systolic array efficiency and systolic cycle length depend on the complexity of the processing element algorithm and the underlying implementation of processing elements. Both bound the systolic array throughput.

Many algorithm post-transformations can be applied to systolic solutions in order to: enhance their effectiveness, lower the number of required processing elements, increase the efficiency, shorten the processing time or increase the throughput. Well known systolic array transformation techniques are: *c*-slowing, folding, double pipelining, fast designs and two-level pipelining [1].

In order to achieve further throughput enhancements, a detailed look within a processing element on the level of the processing element clock cycle is necessary.

2.1. Latencies within systolic arrays

Lower than 100% efficiency. Data flows between processing elements are spaced with dummy values in order to satisfy the requirement for proper data synchronization within a systolic array. Efficiency enhancing techniques outlined above can be used to deal with this problem.

Synchronization problems. There is a number of systolic algorithms where the systolic array is composed of sub-arrays with different types of processing elements. Each sub-array of such systolic array receives its own instruction sequence. Instruction streams must be synchronized.

Data dependencies. Traditionally, non-pipelined functional units were preferred as a building block within a processing element, thus eliminating data hazards. Non-pipelined functional units also exhibit lower latencies in terms of clock cycles and can execute recursions more effectively. With higher clock rates pipelined functional units are preferred, but then we have to cope with data dependency problems.

Long latency of operation with pipelined functional unit.. Simple arithmetic operations (multiply, add, subtract, multiply/accumulate) can be efficiently pipelined. After the initial latency period and if there are enough independent operands to be consecutively supplied to a functional unit, its throughput becomes one datum per a clock cycle. Multiple threads can successfully fill the pipelines of these functional units, if clever instruction scheduling is performed.

Long latency operations with non-pipelined functional unit. Several arithmetic operations can not be efficiently pipelined (at least not with low real estate requirements) as for example division, square root computation, etc. More threads of the same algorithm do provide more independent instructions, but they are serially executed due to the resource constraints.

3. MULTITHREADED SYSTOLIC COMPUTATION

In the proposed approach, multiple independent algorithm threads (i.e. instances of the same algorithm) are interleaved within a given systolic cycle on the same systolic array. Data from multiple threads are pumped through the systolic array in blocks resulting in a dramatic improvement of the net throughput. All threads share the same resources of the systolic array.

We propose employment of pipelined functional units together with a sophisticated compiler/scheduler framework. Multithreading keeps execution pipelines within the processing element full and thus increases the efficiency of functional units. The proposed programmable processing element can exploit simultaneous multithreading of multiple algorithm instances within one systolic cycle.

Each thread can belong to a different systolic algorithm or be another instance of the same systolic algorithm. All threads share the processing element's resources including, the inter-processing element communication data paths. The end result is a shortened algorithm-processing phase achieved by running concurrently interleaved multiple independent algorithm threads on processor arrays. Performance increase is due to the elimination of true data hazards within each algorithm, better functional unit utilisation and larger amounts of usable instruction level parallelism (ILP). Basic block length describing the algorithm within the processing element is lengthened, thus creating more ILP. Functional unit pipelines within processing elements are kept busy by interleaving independent threads running simultaneously through the systolic array. Efficiency of the whole systolic array improves, since many algorithms can finish execution in shorter time than they would when executed serially on the same systolic array.

3.1. Sources of threads within systolic arrays

For multithreaded systolic computation to thrive it is necessary to present multiple threads to the systolic array. There are several sources of data sets that can be treated as unrelated threads:

Data vectors from different partitions. When problems are too big to be fitted directly to a given systolic array, problem partitioning is used to handle processing of sub parts sequentially on systolic array [1].

Loop unrolled systolic cell algorithm. Viewing the execution from the systolic controller's point of view, we can see that processing element computation is executed repetitively in a loop. In each systolic cycle blocks of data, corresponding to unrolled loop instances, are transferred between processing elements.

Multiple instances of the same algorithm. There are some applications that require running the same algorithm several times with different data sources (e.g. channel processing in mobile base stations or xDSL systems). Here different instances of the same algorithm use different filter coefficients and thus have no common points.

Simultaneous execution of different types of algorithms. The goal of processing different types of algorithms simultaneously is to utilize vacant functional units available within processing elements.

Suitable combinations of the above. For example, one algorithm can be unrolled and combined with a more complex counterpart.

3.2. Model of multithreaded systolic computation

We can examine the multithreaded systolic computation on a logical level where all threads are concurrently executing on the systolic array with the granularity of one processing element clock cycle. Implementation of multithreaded programs uses a packet data transfer approach: For each

iteration of the systolic program *M* data elements, one from each of the threads are input, processed and output. A graphical representation of *I/O* activities within a multithreaded systolic array is shown in Figure 2. Data elements of input data vectors of *M* threads are time multiplexed into a multithreaded systolic array at a rate of one element per processing element clock cycle. Data elements of result vectors are output at the same rate. The process repeats for all subsequent elements and all threads.

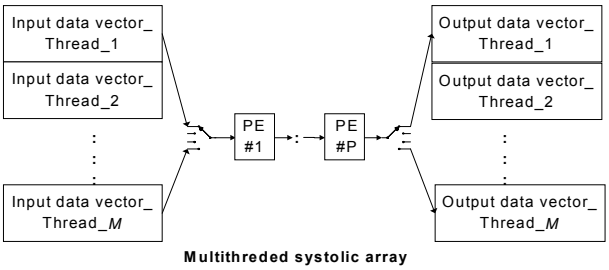

Figure 2: Logical representation of multithreaded systolic computation. Switch rate = 1/PE clock cycle.

Since all functional units within processing elements are pipelined, data are pumped at the pipeline rate of these functional units, compared to the systolic cycle rate in traditional systolic arrays (Figure 1). For each iteration, multiple unrelated data samples from all threads are input, processed and output within one systolic cycle (Figure 3).

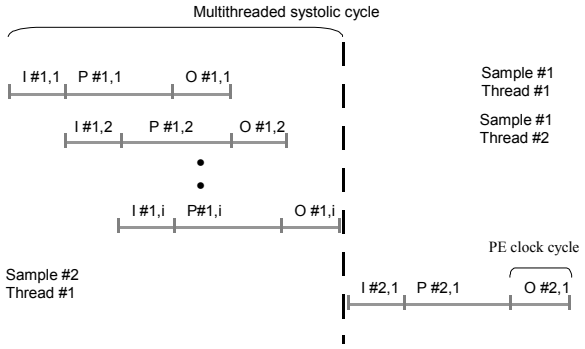

Figure 3: Multithreaded systolic processing phases within a processing element.

Next we identify three function blocks that could limit the performance increase of multithreaded systolic computation:

- *Inter processing element communication channels*. We can expect at some point no increase in the systolic array throughput if bisection bandwidth of the systolic array remains constant.
- *Register file*. Enough operand storage must be provided within each processing element to handle the increased number of data elements from multiple threads.
- *Delay elements*. Delay elements are situated on communications channels and these delays can be implemented within the register file or local memory. The number of delay elements required in multithreaded processing element is the sum of all delays specified for each thread.

4. SIMULATION RESULTS

4.1. Algorithm selection

A basic core of mathematical algorithms arising in problems of modern multimedia and image processing consists of linear algebra and linear operator theory. We focus on systolic algorithms for two well-known linear algebra algorithms, namely Gaussian elimination and Givens rotations [1]. Gaussian elimination is a standard method for solving linear systems of equations and is one of the most widely used algorithms for LU decomposition of matrices [5]. Givens rotations are usually considered as a method for QR decomposition of matrices [5]. More generally, Givens rotations can be used for QR decomposition as well as for orthogonal triangularisation.

4.2. Simulation environment

The simulation is focused on the execution of computations within each processing element of the systolic array. We functionally simulate multithreaded systolic computation operation as it is executed on a processing element employing a parameterised very long instruction word (VLIW) processor simulator.

PE model name	PE operation issue width	Number of functional units by Type (T) / Latency (L)					
		Add		Mul		Div/Sqrt	
		T	L	T	L	T	L
A	5	2	3	2	3	1	17
B	8	2	5	2	5	2	17
C	8	4	5	4	5	4	17

Table 1: Description of processing element models used in the simulations.

Three different models of the processing element are treated in this paper (Table 1). All computational functional units within the three models are pipelined into 5 stages, except for divide/square root units that have 17 processing element clock cycle latency. The number of memory and inter processing element communication functional units is fixed at two for the model A and at eight units for the models B and C respectively and are two stage pipelined. Operation issue width describes the maximum number of operations simultaneously executable within the processing element. There are no restrictions on the type of operations that can proceed concurrently, as long as there are no data or structural hazards.

4.3. Simulation results

The results present speedups achievable on a multithreaded systolic array running the selected algorithms on the presented processing element architectures. Speedups are

based on the execution time ratios of multithreaded systolic computation vs. single threaded systolic computing.

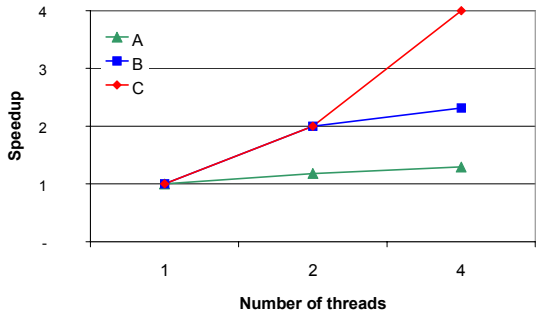

Figure 4: Multithreaded systolic computation of Givens rotations. Speedup.

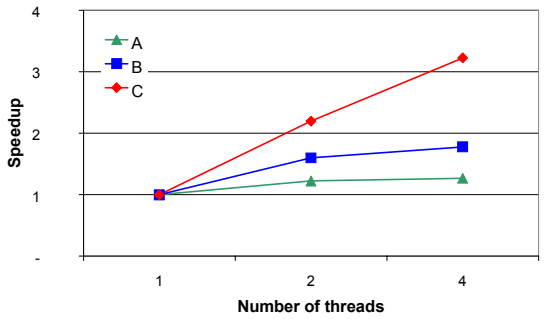

Figure 5: Multithreaded systolic computation of Gaussian elimination. Speedup.

The speedups defined reflect the true processing element clock cycle-by-cycle situation within the processing element and as such they also reflect the net throughput of each processing element. Our definition does not take into account the second level effects as data load/store to/from systolic array.

We can define the speedup S_{MTH} of multithreaded program running one type of algorithm versus single threaded program running the same type of algorithm as

$$SMTH = \frac{\text{Execution time of a single threaded program} \cdot M}{\text{Execution time of multithreaded program}}$$

where M equals to number of threads. Both algorithm versions have to perform an equal job and the equalisation constant is M. Speedup is calculated for the same processing element model running a different number of threads.

From Figure 4 we can observe that the speedup curve flattens already with two threads, since the algorithm contains long latency arithmetic operations, non-pipelineable operations and insufficient number of resources available within a given processing element. A similar observation applies to Figure 5, although higher speedups are achieved, since the long latency operation count is smaller for this algorithm. When multiple long latency functional units are available, than the speedup continues to increase.

Multithreaded systolic computation creates the following effects:

- PE utilization is increased. This is due to the fact that there are many independent operations available and these can be executed concurrently.
- As a direct consequence of multithreading the net throughput increases. As long as there are enough pipelined functional units and inter processing element communication channels, the speedup curve can experience a proportional increase.
- Basic blocks of the code executed on processing elements become longer, which results in the opportunity for the extraction of more ILP.
- Register requirements increase proportionally to the number of threads.

5. CONCLUSIONS

Multithreaded systolic computation results in higher systolic array's throughput, due to improved utilization of pipelined functional units within processing elements.

A linear increase in the throughput is observed as long as the processing element functional units are pipelined and the algorithms do not experience very low computation-to-communication ratios. In the case of linear algebra algorithms that require the computation of lengthy square root and division operations that can not be pipelined, multithreading can not provide any significant benefit unless multiple functional units are employed. Classical DSP algorithms require only multiply-accumulate type of operations and as such they can experience even better speedup results.

Programmability of individual processors of the systolic array is a key to the straightforward implementation of various algorithms with a multithreaded systolic computation concept. Multithreaded systolic computation provides speedups that asymptotically approach the number of threads executed simultaneously on the systolic array. The ideas outlined in the paper are equally well applicable to systolic and SIMD arrays alike.

6. REFERENCES

[1] High Performance VLSI Signal Processing: Innovative Architectures and Algorithms, vol. I, II, Edited by: K. J. R. Liu, K. Yao, IEEE Press, 1998.

[2] Special issue on: Systolic Arrays, Computer, vol. 20, no. 7, July 1987.

[3] R. Sernec, M. Zajc, "Multithreaded systolic computation: A novel approach to performance enhancement of systolic arrays", *Elektrotehniški vestnik*, vol. 68 (2-3): 81- 89, 2001.

[4] Special issue on: The Future of Micro Processors, Computer, vol. 30, no. 9, September 1997.

[5] G. H. Golub, C. F. Loan, Matrix Computations, 3. edition, Johns Hopkins, 1996.