CORE
Metadata, citation and similar papers at core.ac.uk
Provided by Loughborough University Institutional Repository

Loughborough University

This item was submitted to Loughborough's Institutional Repository (https://dspace.lboro.ac.uk/) by the author and is made available under the following Creative Commons Licence conditions.

creative commons
COMMONS DEED

Attribution-NonCommercial-NoDerivs 2.5

You are free:

- to copy, distribute, display, and perform the work

Under the following conditions:

Attribution. You must attribute the work in the manner specified by the author or licensor.

Noncommercial. You may not use this work for commercial purposes.

No Derivative Works. You may not alter, transform, or build upon this work.

- For any reuse or distribution, you must make clear to others the license terms of this work.
- Any of these conditions can be waived if you get permission from the copyright holder.

Your fair use and other rights are in no way affected by the above.

This is a human-readable summary of the Legal Code (the full license).

Disclaimer

For the full text of this licence, please go to:
http://creativecommons.org/licenses/by-nc-nd/2.5/

Parametric Data-Parallel Architectures for TLM Acceleration

Vassilios A Chouliaras James A Flint Yibin Li
V.A.Chouliaras@lboro.ac.uk J.A.Flint@lboro.ac.uk Y.Li2@lboro.ac.uk

Department of Electronic and Electrical Engineering, Loughborough University,
Ashby Road, Loughborough, Leicestershire, LE11 3TU, UK.
Tel: +44 1509 227113, Fax: +44 1509 227014

Abstract– **We discuss the architecture and microarchitecture of a scalable, parametric vector accelerator for the TLM algorithm. Architecture-level experimentation demonstrates an order of magnitude complexity reduction for vector lengths of 16 32-bit single-precision elements. We envisage the proposed architecture replicated in a SoC environment thus, forming a multiprocessor system capable of tapping parallelism at the thread level as well as the data level.**

1. Introduction

Prior attempts to implement the TLM algorithm [1] on general-purpose architectures have fallen into two major categories: Shared memory, cache coherent multi-processors [2, 3] and distributed processors [4] with shared- memory machines often demonstrating better performance.
The TLM is a highly-parallel three-dimensional numerical algorithm which has the potential for being accelerated along its innermost loop via vectorization thus, tapping parallelism at the data level (DLP). Furthermore, the algorithm can be statically 'sliced' (threaded) along the second outer loop, and be executed on the previously mentioned platforms via different processors executing different iterations. Such parallelism is known as thread-level-parallelism [5] and is currently being pursued by all major microprocessor vendors.
Successful acceleration of such parallel codes depends very much on the algorithmic communication pattern which dictates the level of data sharing across the multiple processors. In the case of the TLM, data transfers between individual nodes is very high and in extreme cases the data transfer during the *connect* part of the algorithm can be much more computationally expensive than the numerical calculations during *scattering*. The performance differential between shared memory and distributed machines is often attributed to such data sharing issues.
Custom architectures for accelerating TLM codes have been proposed in the past by Stothard and Pomeroy [6]. Our work proposes a custom vector approach to accelerating the inner loop of TLM codes, quite unlike this earlier work. In our case, an embedded 32-bit processor is augmented with a configurable, extensible custom vector accelerator and resides on an on-chip-bus [7] thus, forming a finely-tuned SoC computation kernel for the TLM algorithm.

2. Vector architecture

The programmer's model of the parametric vector accelerator for TLM is depicted in Figure 1:

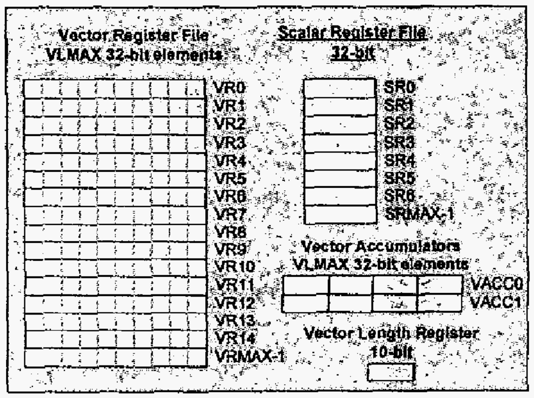

Figure 1: Vector accelerator programmer's model

The programmer's model specifies a parametric number of vector registers (VRMAX), each consisting of a parametric number of 32-bit single-precision elements (VLMAX). There is a scalar register file consisting of a parametric number of scalar 32-bit elements (SRMAX), used for virtual address computation, immediate passing and vector splat operations. Additionally, there are two vector accumulators each holding VLMAX single-precision elements and finally, the vector length register (VLEN) which specifies the number of bytes that will be affected by the currently executing vector opcode.
The Instruction Set Architecture (ISA) of the accelerator includes standard vector floating point operations except division, vector Load/Stores, and a generalized permute instruction. A large number of sub-element manipulation instructions (including vector splat instructions) can be synthesized based on the three-operand permute infrastructure. The ISA is summarized in Table 1

Authorized licensed use limited to: LOUGHBOROUGH UNIVERSITY. Downloaded on May 05,2010 at 15:48:02 UTC from IEEE Xplore. Restrictions apply.

Table 1: Vector Coprocessor ISA

Instruction	*Description*
MVSR2VLEN	Transfer scalar register to vector length register (VLEN)
MVSR2CSR	Transfer RISC scalar register to coprocessor scalar register
MVCSR2R	Transfer coprocessor scalar register to RISC register
MVSR2CVEL	Move RISC scalar register to coprocessor vector element
MVCVEL2R	Move coprocessor vector element to RISC scalar register
VLDU	Load vector register unaligned under VLEN
VSTU	Store vector register unaligned under VLEN
VPERM	Three-operand bytewise vector permute
VSPLAT	Splat coprocessor scalar register to coprocessor vector register
VFPADD.S	Vector floating-point add (single precision) under VLEN
VFPSUB.S	Vector floating-point sub (single precision) under VLEN
VFPMUL.S	Vector floating-point mult (single precision) under VLEN
VFPMAC.S	Vector floating-point multiply-accumulate

3. Vector microarchitecture

The proposed vector extensions are implemented as a tightly-coupled vector accelerator attached to an open-source, configurable, extensible, Sparc V8-compliant RISC CPU [8]. The processor/coprocessor combination communicates via the AHB On-Chip Bus to the SDRAM controller which controls the off-chip SDRAM part. A high level schematic of the scalar processor and vector accelerator is depicted in Figure 2.

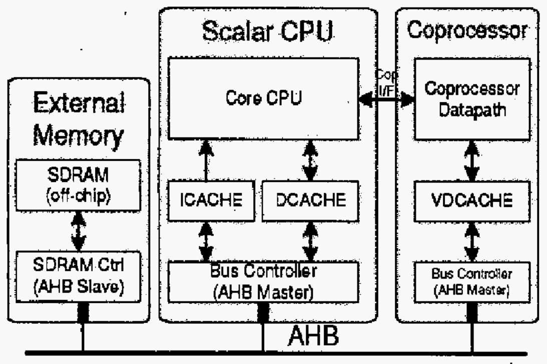

Figure 2: TLM computation kernel

As shown in the figure, there exists a bidirectional communication channel across the scalar processor and the vector accelerator. Though the open source CPU provides a coprocessor interface, it was decided to implement that channel in order to ensure pipelined, lockstep operation of the accelerator and timely transfer of data to and from the main CPU. Typical transactions on the developed channel are depicted in Figure 3.

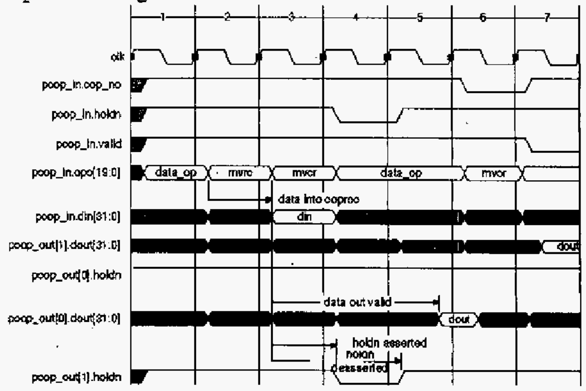

Figure 3: Processor-Coprocessor communication channel

The detailed microarchitecture of the combined scalar processor/vector coprocessor for a vector length of two 32-bit (single-precision) elements is depicted in Figure 4.

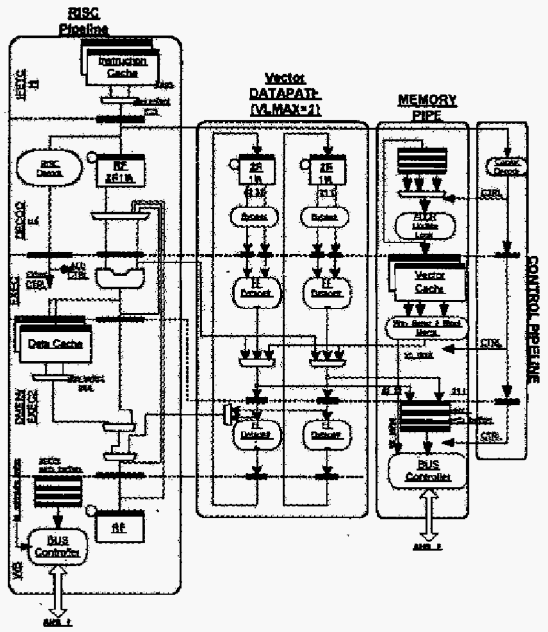

Figure 4: Detailed microarchitecture

Instructions are fetched from the multi-way set-associative instruction cache and stored in a single 32-bit register. Typically, high-performance RISC processors of equal pipeline depth would extract the source operand fields right after instruction cache access and set up the synchronous register file. Unfortunately, that is not the case in the particular processor which, due to the windowing scheme of the Sparc V8 architecture, requires access to the current-

Authorized licensed use limited to: LOUGHBOROUGH UNIVERSITY. Downloaded on May 05,2010 at 15:48:02 UTC from IEEE Xplore. Restrictions apply.

window-pointer (CWP) register in order to compute a physical register file address. As a result, source operand addresses are set-up on the falling edge of the clock in the DECODE stage. During this stage, the register file is accessed and the two source registers are retrieved. Operand bypassing takes then place and the resolved operands are clocked into the ALU input registers, ready for execution. It is during this stage that the vector opcodes are identified and dispatched to the tightly-coupled vector accelerator. Decoding logic in the later produces a number of control fields which are pipelined down the control pipeline. Vector operand accesses are triggered by the falling edge of the clock during decode, for reasons of symmetry to the scalar pipeline.

During the EXEC stage, the RISC CPU executes the scalar instruction or computes the virtual address of a Load/Store operation. In the same stage, the vector accelerator performs the first stage of the pipelined floating point computations. In the next stage, scalar data return to the main processor via the data cache return path whereas the vector accelerator performs the last stage of execution. Due to the very tight timing constraints, floating point results are stored in an intermediate register prior to committing to the vector register file.

4. Methodology

We have applied a basic implementation of the SCN TLM algorithm [1] in which no external boundary conditions were used. In the particular case, a single output node was used as a diagnostic aid to verify correct operation. We used the accelerated scatter method of Naylor and Ait-Sadi as proposed in [9].

The non-vectorized (scalar) algorithm was profiled both in native mode (IA32 Linux) as well as on our simulated processor for consistency of results. Scalar code profiling revealed a scatter:connect complexity ratio of 63:37, averaging over all the studied configurations.

Our simulation infrastructure is based around the simplescalar toolset [10] which provides a complete computer architecture modelling and performance evaluation environment. The compiler used was GCC 2.7.3 with optimizations (-O3).

5. Results

The reference problem chosen for benchmarking was a fixed mesh of 10^6 nodes. This number is convenient as it gives a prime factorisation of $2^6 \times 5^6$, which allows for the aspect ratio of the problem space to be varied over a reasonable range whilst maintaining the same number of nodes.

We measured the absolute complexity (dynamic instruction count) of the scalar code for all configurations of interest. Then, the vectorized code was run and its complexity recorded for a maximum vector length of up to 16 single-precision elements. Figures 5 and 6 depict the normalized complexity of the vectorized algorithm over maximum vector length.

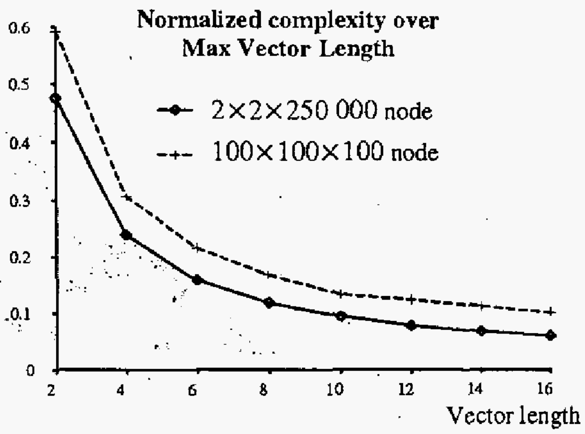

Figure 5 – Benchmarking using a thin and a cubic problem space.

Figure 5 suggests that the optimal (less complex) configuration is where the problem space is thin, i.e. where the vector length is maximised.

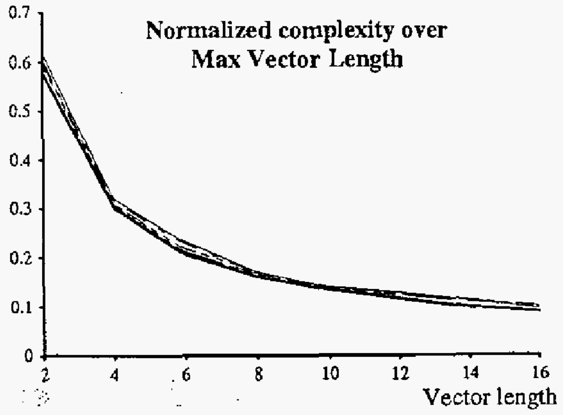

Figure 6 – Iteration time for 80 x 100 x 125 node mesh with differing alignment relative to the vector direction. All of these results show a similar speedup

Figure 6 depicts a 80×100×125 configuration compared with a mesh of 100×125×80, 80×125×100, etc. These mesh dimensions were chosen as being typical of realistic model of an electromagnetic scattering situation. Results demonstrate that vectorization alignment changes only slightly the complexity (and hence run time) in all configurations. A vector length of 16 single-precision elements showed a speedup of approximately an order of magnitude thus clearly demonstrating the benefit of using parallelism at the data level.

Authorized licensed use limited to: LOUGHBOROUGH UNIVERSITY. Downloaded on May 05,2010 at 15:48:02 UTC from IEEE Xplore. Restrictions apply.

6. Conclusions

We have proposed a parametric vector accelerator to exploit the significant amount of data level parallelism which is inherent within the TLM code. Our results demonstrate an order-of-magnitude performance improvement can be achieved for a vector length of 16 single-precision elements. Such a configuration is realizable with current VLSI technology. We are also actively investigating thread-level parallelism as the second major source of parallelism in the workload. Our scalable architecture can be replicated thus, creating a cache-coherent, embedded multiprocessor for TLM acceleration providing further performance benefits.

References

[1] P. B. Johns, *"A symmetrical condensed node for the TLM method"*, IEEE Trans. Microw. Theory Tech., vol. 35, no. 4, pp. 370–377, 1987.

[2] J. L. Dubard, O. Benevello, D. Pompei, J. Le Roux, P. P. M. So, and W. J. R. Hoefer, *"Acceleration of TLM through signal processing and parallel computing"*, in Computation in Electromagnetics, pp. 71–74, IEE, 25–27 November 1991.

[3] C. C. Tan and V. F. Fusco, *"TLM modelling using an SIMD computer"*, Int. J. Numerical Modelling: Electronic Networks, Devices and Fields, vol. 6, pp. 299–304, 1993.

[4] P. J. Parsons, S. R. Jaques, S. H. Pulko, and F. A. Rabhi, *"TLM modeling using distributed computing"*, IEEE Microw. and Guided Wave Lett., vol. 6, no. 3, pp. 141–142, 1996.

[5] J. Henessy, D. A. Patterson *"Computer architecture: A quantitative approach"*, Morgan Kaufmann publishers, ISBN 1-55860-329-8

[6] D. Stothard and S. C. Pomeroy, *"Dedicated TLM array processor"*, Applied Computational Electromagn. Soc. J., vol. 13, no. 2, pp. 188–196, 1998.

[7] *"AMBA Specification (Rev 2.0)"*, www.arm.com

[8] *"The Leon-2 processor User's manual, XST edition, ver. 1.0.14"*, http://www.gaisler.com

[9] P. Naylor and R. Ait-Sadi, *"Simple method for determining 3-D TLM nodal scattering in nonscalar problems"*, Electron. Lett., vol. 28, no. 25, pp. 2353–2354, 1992.

[10] D. Burger, T. Austin, *'Evaluating Future Microprocessors: The Simplescalar Tool Set'*, http://www.simplescalar.com

Authorized licensed use limited to: LOUGHBOROUGH UNIVERSITY. Downloaded on May 05,2010 at 15:48:02 UTC from IEEE Xplore. Restrictions apply.