PARALLEL PROGRAMMING OF A SYMMETRIC TRANSPORT-TRIGGERED ARCHITECTURE WITH APPLICATIONS IN FLEXIBLE LDPC ENCODING

Blaine Rister[⋆]*, Pekka Jääskeläinen*[†]*, Olli Silvén*[‡]*, Jari Hannuksela*[‡]*, and Joseph R. Cavallaro*[⋆]

[⋆] Department of Electrical and Computer Engineering, Rice University, Houston, Texas
blaine.rister@rice.edu, cavallar@rice.edu
[†] Department of Pervasive Computing, Tampere University of Technology, Tampere, Finland
pekka.jaaskelainen@tut.fi
[‡] Department of Computer Science and Engineering, University of Oulu, Oulu, Finland
olli.silven@oulu.fi, jhannuks@ee.oulu.fi

ABSTRACT

Exposed-datapath architectures yield small, low-power processors that trade instruction word length for aggressive compile-time scheduling and a high degree of instruction-level parallelism. In this paper, we present a general-purpose parallel accelerator consisting of a main processor and eight symmetric clusters, all in a single core. Use of a lightweight and memory-efficient application programming interface allows for the first high-performance program executing both sequential and data-parallel code on the same TTA processor. We use the processor for LDPC encoding, a popular method of forward error correction. Demonstrating the flexibility of software-defined radio, we benchmark the processor with two programs, one which can handle almost any sort of LDPC code, and another which is optimized for a specific standard. We achieve a throughput of 5 Mb/s with the flexible program and 92 Mb/s with the standard-specific one, while consuming only 95 mW at a clock frequency of 1175 MHz.

Index Terms— parallel computing, Open Computing Language (OpenCL), transport-triggered architecture (TTA), software-defined radio (SDR), low-density parity check codes (LDPC)

1. INTRODUCTION

As the era of Moore's law draws to a close, we must look for novel ways to accelerate parallel workloads, especially within the power constraints of embedded systems. We are witnessing a significant increase in the number and diversity of processors used to realize the computational needs of today's products [1]. For example, contemporary smartphone and tablet system-on-chip (SoC) designs feature CPUs, special-purpose processors such as graphics processing units (GPUs), digital signal processors (DSPs), and fixed-function accelerators for communications, video, and audio processing [2]. We propose a TTA processor that exploits instruction-level parallelism (ILP) in sequential code, as well as data parallelism in parallel programs, meant to serve as a co-processor in such systems. Although it is suited for general-purpose parallel workloads, we demonstrate the processor's capabilities in a communications role.

LDPC codes are a popular method of capacity-approaching forward error correction (FEC) that have been adopted in standards such as IEEE 802.11n and DVB-S2 [3]. We desire to use flexible LDPC encoding on battery-powered platforms transmitting a large amount of data wirelessly, such as portable medical imaging devices. TTA architectures have already been designed for FEC decoding, but for our application data transmission takes precedence over reception, and flexibility poses a more challenging encoding problem [4, 5]. FEC schemes are usually implemented with fixed-function hardware accelerators, but this results in slow time-to-market, inflexible designs, and may consume a large area when many different accelerators are needed on the same platform. As an alternative, we implement LDPC encoding in software on our processor. Generalized LDPC encoding is a fitting demonstration of our processor's capabilities because it involves common, parallelizable operations in sparse and dense linear algebra. We use no application-specific hardware, maintaining processor flexibility.

To our knowledge, this is the first implementation of LDPC encoding on a TTA processor. Additionally, we have improved on existing implementations of vector TTA processors and the OpenCL parallel computing standard so that we may efficiently run sequential C and parallel OpenCL code on the same core, avoiding the commonly observed communication overhead across cores and address spaces [6]. We have carefully constructed OpenCL kernels which process sparse and dense matrices with improved parallelism. We test our processor on a general algorithm that is flexible across many LDPC codes, as well as a standard-specific algorithm that achieves higher throughput.

978-1-4799-2893-4/14/$31.00 ©2014 IEEE

2. OVERVIEW OF LDPC ENCODING

Compared to turbo codes, LDPC codes admit considerably higher encoding complexity [7, 8]. A binary LDPC code is represented by a sparse parity check matrix H, which encodes a set of homogeneous linear equations [9]. The encoding problem is to find the codeword satisfying the system for each source vector. While it is possible to formulate a generator matrix and encode by dense matrix multiplication, a more efficient algorithm was found using mostly sparse operations [7]. The algorithm works on all codes which are in approximate lower-triangular (ALT) form, meaning H can be made lower-triangular by removing its last g rows, where g is called the *gap* of the code. Partitioning H by the gap once along the rows, and once along the columns, we are left with six sub-matrices:

$$H_{m\times n} = \begin{pmatrix} A_{(m-g)\times(n-m)} & B_{(m-g)\times g} & T_{(m-g)\times(m-g)} \\ C_{g\times(n-m)} & D_{g\times g} & E_{g\times(m-g)} \end{pmatrix}$$

2.1. General encoding

The algorithm has two phases: preprocessing for a given code, and encoding each packet. In the preprocessing phase, we compute the matrix $\phi^{-1} = (-ET^{-1}B + D)^{-1}$. In the encoding phase, we form a packet $x = (s, p_1, p_2)$, where s is the source vector and (p_1, p_2) are the parity bits. The parity check equation $Hx^T = 0^T$ is equivalent to the following system:

$$As^T + Bp_1^T + Tp_2^T = 0 \quad (1)$$

$$(-ET^{-1}A + C)s^T + \phi p_1^T = 0 \quad (2)$$

The solutions for the parity bits are as follows:

$$p_1^T = -\phi^{-1}(-ET^{-1}As^T + Cs^T)$$
$$p_2^T = -T^{-1}(As^T + Bp_1^T)$$

Since T is lower-triangular, multiplication by T^{-1} may be efficiently solved by forward substitution, and in the finite field $GF(2)$, we ignore the negations [10]. For each packet, the algorithm requires five sparse matrix-vector multiplications, one dense matrix-vector multiplication, two dense vector additions, and two sparse forward substitutions.

2.2. Structured codes

To further simplify the encoding problem, it is common to use codes with special repetitive structures [11]. For example, *quasi-cyclic LDPC codes* (QC-LDPC) consist of cyclic permutation matrices of a regular size z. For these codes, matrix-vector multiplication is equivalent to the accumulation of rotated sub-vectors of the vector multiplicand. It is also common to choose codes such that ϕ is the identity matrix, avoiding dense matrix multiplication entirely. In this paper, we first demonstrate the flexibility of SDR on a general encoding program, capable of supporting many different standards, and then demonstrate higher throughput with a program tailored to a specific structured code.

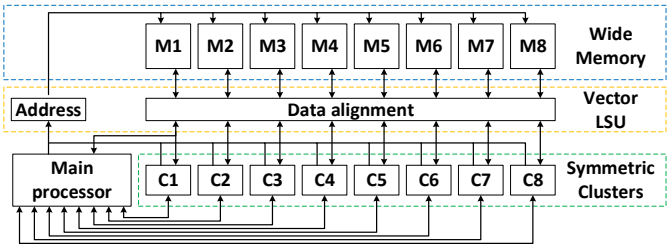

Fig. 1. Processor architecture with vector data memory access.

3. PROCESSOR ARCHITECTURE

As with other very long instruction word (VLIW) processors, TTA processors allow for instruction-level parallelism by encoding multiple parallel operations into a single processor instruction. TTA takes this approach a step further by encoding the movement of data between function units (FUs) into the instruction, affording numerous advantages in performance and hardware complexity [10, 12]. Firstly, the compile-time scheduler is able to reduce latency and register file pressure by routing data from the output of one FU to the input of the next, skipping the register file in what is called *software bypassing*. Secondly, datapath exposure enables the use of a large number of small register files with few ports, drastically reducing their complexity. Thirdly, elimination of runtime control tasks such as register renaming results in small, power-efficient processors [12].

3.1. Symmetric clusters for parallel code

Our processor adds data parallelism to TTA by including eight symmetric clusters and a vector load-store unit (LSU) for wide memory access, as seen in figure 1. We can load eight 32-bit words from data memory in parallel, and process them in the clusters. The LSU can also be used for scalar accesses down to the byte level. The results can be stored in parallel, or quickly transferred to the main processor for later use. The clusters need not be confined to the single instruction, multiple data (SIMD) model when it is more efficient to schedule them sequentially, or to use different operations in each cluster. Eight transport buses allow data to be transferred from cluster registers to main processor registers.

In the past, vector TTA processors used interleaved memory, which increases latency and hardware complexity [4, 13]. We have found that many signal processing applications can be written to use linear memory access, so the wide 256-bit data words of our simpler memory hierarchy are a fast and cost-effective solution. Another weakness of previous designs was that clusters could not send addresses to the vector LSU,

Table 1. Function units and buses of the processor. Multiply cluster FUs by 8 for total.

Resource	ALU	Mult.	FPU	RF*	Bus†
Cluster	2	1	1	3	3
Main proc.	6	4	4	9	6
Total	22	12	12	33	38‡

*The largest register file has three 32-bit registers.
†We can perform one parallel transport operation for each bus.
‡This includes 8 transport buses.

resulting in an inefficient schedule. We solve this problem by connecting all clusters to the 32-bit serial read and write ports on the LSU, as well as the triggering address port.

3.2. Main processor for sequential code

A few vector TTA processors have been proposed, but ours differs in that it is also meant to process sequential code through the larger main processor. In past designs, it was assumed that a PCI Express bus could transport data to an applications processor when sequential execution was necessary [4]. In our previous work, we have found that transferring data between parallel and sequential processors was a significant bottleneck, particularly in heterogeneous CPU and GPU systems [6, 14]. Our design solves this problem by executing both types of code on the same chip, in the same address space, reducing data transfer overheads between sequential and parallel code.

Both the clusters and the main processor can trigger multiple FUs from table 1 in parallel, while pipelining operations to the multipliers, arithmetic and logic units (ALUs), and floating point units (FPUs). For a more flexible architecture, we include parallel 32-bit FPUs to support future applications. If the need arises, the modular nature of TTA allows us to quickly add or remove hardware from existing designs.

4. APPLICATION CODE AND OPENCL API IMPLEMENTATION

In this section, we describe how our efficient application code was tailored to our architecture and compiler. We also describe our implementation of the OpenCL application programming interface (API) for TTA processors, which makes efficient use of our memory model. For our application, we wish to support a wide range of codes, so we provide more explanation for the general encoding algorithm, but we briefly describe the necessary modifications to optimize for a common standard. We provide some initial performance analysis to show the benefits of our optimizations. Our testing methodology is described in §5.2.

Program 1 OpenCL kernel for binary sparse matrix-vector multiplication, called for each matrix element.

```
kernel void multiply(global const int *mat,
const int num_rows, global const uchar *v_in,
global uchar *v_out) {
  int x, y, id;
  x = get_global_id(1); y = get_global_id(0);
  idx = mat[y + x * num_rows];
  atomic_xor((global uint *) (v_out + y),
  (uint) v_in[idx] & (idx >= 0)); }
```

4.1. Sparse matrix-vector multiplication

Our tests show that sparse matrix-vector multiplication is the most expensive operation in LDPC encoding, accounting for 73 percent of the total execution time after optimization. We surveyed existing methods developed for GPUs, but they involved loops and conditionals [15]. These are problematic for our architecture, which controls all clusters on the same core. Instead, we choose as our OpenCL work item a single element of the matrix, using a two-dimensional NDRange, and accumulating the partial products in parallel, as seen in program 1. Binary sparse matrices are encoded as an array of nonzero indices. We use the ELLPACK sparse matrix format because it preserves row position [15]. The number -1 is used to encode empty elements in the sparse matrix, so we mask away these elements with the operation & (idx > 0), where idx is the index retrieved from the sparse matrix. Finally, all matrices are stored in column-major order so that eight elements in a column may be fetched in parallel as linear memory, their results stored in parallel. The program alternates between parallel linear memory accesses and sequential random accesses, in which non-loading clusters may perform other work, demonstrating multiple instruction, multiple data (MIMD) freedom in workload partitioning.

4.2. Other operations

We implement vector addition and dense matrix multiplication in the same style, this time using the column-major dense matrix format. For these operations, four *unsigned char* data types are loaded as a single 32-bit *unsigned int*, reducing the number of required memory accesses by a factor of four. Zero-padding is used to avoid checking matrix boundaries. With vector accesses, we may load 32 packed *unsigned char* values at once. This technique accelerates dense matrix multiplication by a factor of 2.5.

Sparse forward substitution exhibits a high degree of data dependency, making it a poor choice for parallelization. We implement this stage in manually-unrolled C code, encoding the XOR operations at compile time instead of accessing the sub-matrix T, and improving opportunities for ILP. On our processor, manual unrolling accelerates forward substitution

by a factor of 16.5 over a naive implementation.

4.3. Standard-specific methods

We also test an alternative algorithm optimized for the IEEE 802.11n standard with a sub-block width of 27 bits. We do not include standard-specific hardware, demonstrating the flexibility of the processor by implementing these optimizations using the existing FUs.

QC-LDPC codes accelerate matrix-vector multiplication, as explained in §2.2. We use a dense matrix to encode sub-block rotations, encoding empty sub-blocks with -1, as in §4.1. We multiply by A using an OpenCL kernel similar to program 1. Because the other operations are inexpensive for this QC-LDPC code, they do not warrant the overhead of launching OpenCL kernels. Additional optimizations were made to forward substitution based on the staircase structure of the T sub-matrix. Finally, for these codes, ϕ is the identity matrix, so we avoid dense matrix-vector multiplication.

4.4. Efficient OpenCL implementation

OpenCL is a cross-platform standard for parallel programming [16]. Targeting the clusters with parallel program kernels allows for easier alias analysis and workload partitioning [17]. However, control code for the OpenCL kernels proved to be very expensive on our statically-scheduled architecture. To solve this problem, we eliminated error checking in common subroutines and omitted redundant API functionality.

OpenCL usually calls for separate private, local, and global address spaces, each addressable from different sets of parallel processing elements [16]. Our processor is designed to switch between serial and parallel execution frequently, so we store all data in the same physical address space. Every OpenCL buffer is initialized with an array created in C, and when a call is made to read a buffer into its initializing array, the API copies nothing. In this way, we tremendously reduce the cost of switching between C and OpenCL code.

5. EXPERIMENTAL RESULTS

In this section, we report area, power, and timing estimates for the hardware, as well as throughput for the software, and compare to implementations on field programmable gate arrays (FPGAs).

5.1. Hardware statistics

We obtained power, area, and timing estimates from Synopsis Design Compiler E-2010 using 45nm technology. The processor core and memory interface were synthesized from VHDL. Because of the wide variation among possible data and instruction memory hierarchies, which depend on the applications desired, we do not report their cost. The processor consists of 323×10^3 gates, consuming an estimated 95 mW at 1175 MHz. This is equivalent to .08 mW/MHz, far lower than contemporary smartphone CPUs and GPUs, for which we might expect .5 mW/MHz [6, 18]. The simplified control hardware of TTA is both area- and power-efficient.

Table 2. Comparison of LDPC encoder implementations.

Technology	Method	Length	Gap	Mb/s*
Prop. TTA 1175Mhz	Flexible	648†	27†	5
Prop. TTA 1175Mhz	Specific	648†	27†	92
FPGA 200 MHz [19]	Flexible	9612	182	17
FPGA 143 MHz [8]	Flexible	2000	2	22

*Here we report the number of source bits processed per second.
†Our code comes from the IEEE 802.11n standard.

5.2. Software simulation

Software results come from an instruction cycle-accurate simulator [10]. Throughput calculations assume data and instructions may be fetched from static memory. We compare our processor running both the flexible and standard-specific encoding programs against FPGA implementations from the literature, as shown in table 2. FPGAs are a popular platform for communications processing, and are the most significant rival to application-specific integrated processors (ASIPs) in reconfigurability and performance. Due to the vast number of proposed LDPC codes, it is difficult to compare encoder implementations, so we focused on flexible encoders tested on codes of rate $\frac{1}{2}$. Even still, differences in code gap size affect performance considerably. We achieve a sizable fraction of the throughput of FPGAs using a flexible processor that meets the power and area constraints of embedded systems, and 5 Mb/s is more than enough for our sensor applications. We also observe a throughput of 92 Mb/s with the standard-specific program, satisfying wireless transmission needs for a wider range of applications.

6. CONCLUSIONS

We presented a flexible, power- and area-efficient processor capable of accelerating a wide variety of parallel applications, and explained our techniques for generating efficient system- and application-level software for OpenCL-based LDPC encoding. We have shown that the processor is capable of meeting our performance requirements on two different programs without application-specific hardware.

Acknowledgements

This work was supported by the Finnish Funding Agency for Technology and Innovation under the Parallel Acceleration project, funding decision 40115/13, the Academy of Finland under funding decision 253087, and the United States National Science Foundation under grants CNS-1265332 and ECCS-1232274.

7. REFERENCES

[1] A. Munshi, B. Gaster, T. G. Mattson, J. Fung, and D. Ginsburg, *OpenCL Programming Guide*, Addison-Wesley, 2011.

[2] Qualcomm Inc., *Qualcomm Snapdragon Processor*, http://www.qualcomm.com/chipsets/snapdragon.

[3] D. J C MacKay and R. M. Neal, "Near shannon limit performance of low density parity check codes," *Electronics Letters*, vol. 33, no. 6, pp. 457–458, 1997.

[4] H. Kultala, O. Esko, P. Jaaskelainen, V. Guzma, J. Takala, Jiao Xianjun, T. Zetterman, and H. Berg, "Turbo decoding on tailored opencl processor," in *9th International Wireless Communications and Mobile Computing Conference (IWCMC)*, 2013, pp. 1095–1100.

[5] S. Shahabuddian, J. Janhunen, and M. Juntti, "Design of a transport triggered architecture processor for flexible iterative turbo decoder," in *Wireless Innovation Forum Conference on Wireless Communications Technologies and Software Defined Radio (SDR WInnComm)*, January 2013.

[6] Guohui Wang, Blaine Rister, and Joseph R Cavallaro, "Workload analysis and efficient opencl-based implementation of SIFT algorithm on a smartphone," in *IEEE Global Conference on Signal and Information Processing (GlobalSIP)*. December 2013, IEEE Signal Processing Society.

[7] T.J. Richardson and R.L. Urbanke, "Efficient encoding of low-density parity-check codes," *IEEE Transactions on Information Theory*, vol. 47, no. 2, pp. 638–656, 2001.

[8] D.-U. Lee, W. Luk, C. Wang, and C. Jones, "A flexible hardware encoder for low-density parity-check codes," in *12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2004, pp. 101–111.

[9] Robert G. Gallager, "Low-density parity-check codes," 1963.

[10] Otto Esko, Pekka Jääskeläinen, Pablo Huerta, Carlos S. de La Lama, Jarmo Takala, and Jose Ignacio Martinez, "Customized exposed datapath soft-core design flow with compiler support," in *Proceedings of the 2010 International Conference on Field Programmable Logic and Applications*, Washington, DC, USA, 2010, FPL '10, pp. 217–222, IEEE Computer Society.

[11] Seho Myung, Kyeongcheol Yang, and Jaeyoel Kim, "Quasi-cyclic LDPC codes for fast encoding," *IEEE Transactions on Information Theory*, vol. 51, no. 8, pp. 2894–2901, 2005.

[12] Henk Corporaal, "Transport triggered architectures examined for general purpose applications," in *In Sixth WorkshopComputer Systems, Delft*, 1993, pp. 55–71.

[13] Jarno K. Tanskanen, Teemu Pitkänen, Risto Mäkinen, and Jarmo Takala, "Parallel memory architecture for TTA processor.," in *SAMOS*. 2007, vol. 4599 of *Lecture Notes in Computer Science*, pp. 273–282, Springer.

[14] Blaine Rister, Guohui Wang, Michael Wu, and Joseph R Cavallaro, "A fast and efficient SIFT detector using the mobile GPU," in *IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, May 2013.

[15] Nathan Bell and Michael Garland, "Efficient sparse matrix-vector multiplication on CUDA," NVIDIA Technical Report NVR-2008-004, NVIDIA Corporation, December 2008.

[16] The Khronos Group, *The OpenCL Specification*.

[17] P.O. Jääskeläinen, C.S. de la Lama, P. Huerta, and J.H. Takala, "Opencl-based design methodology for application-specific processors," in *2010 International Conference on Embedded Computer Systems (SAMOS)*, 2010, pp. 223–230.

[18] Xiang Chen, Yiran Chen, Zhan Ma, and Felix C. A. Fernandes, "How is energy consumed in smartphone display applications?," in *14th Workshop on Mobile Computing Systems and Applications*, New York, NY, USA, 2013, pp. 3:1–3:6, ACM.

[19] XiangRan Sun, Zhibin Zeng, and Zhanxin Yang, "A novel low complexity LDPC encoder based on optimized RU algorithm with backtracking," in *International Conference on Multimedia Technology (ICMT)*, 2010, pp. 1–4.