# A Software LDPC Decoder Implemented on a Many-Core Array of Programmable Processors

Brent Bohnenstiehl and Bevan Baas
Department of Electrical and Computer Engineering
University of California, Davis
{bvbohnen, bbaas}@ucdavis.edu

***Abstract*—This paper presents the design and implementation of a software Low Density Parity Check (LDPC) decoder on the AsAP2 platform, which contains a 2D mesh of 164 programmable processors designed for general DSP applications. A software decoding algorithm is described which requires low memory overhead, and scalable methods are provided for parallelizing the computational workload across many cores. LDPC codes of length 4095 and 16129 are implemented, respectively using 152 or 156 processors, achieving 21.3 or 13.4 Mbps of throughput, and using 131 or 188 nJ per decoded bit over four decoding iterations.**

## I. Introduction

Low Density Parity Check (LDPC) codes, introduced in 1962 by Gallager [1], have become increasingly popular in recent years, showing up in multiple communication protocols such as 802.3an (Ethernet), 802.16e (WiMAX), and 802.11n (Wi-Fi). Development of LDPC hardware decoders involve high design costs and time, and such decoders tend to have limited capability to adapt to alternate LDPC codes, leading to software decoder implementations as an attractive alternative.

Interest continues to grow in many-core designs as a means of effectively utilizing the increasing number available transistors on a chip. One thrust of this effort has been focused on high numbers of small, software programmable processors [2], [3], leading to potentially thousands of such processors operating independently on a single die. These designs offer potentially high performance at low power for DSP applications, and are a promising platform for software-based LDPC decoding.

This work focuses on one such platform, the second generation Asynchronous Array of Processors (AsAP2) introduced by Truong et al. [2]. A software decoding algorithm is described in section II. Section III describes two implementations of the algorithm on AsAP2 for sample LDPC codes of different lengths. Section IV presents the final designs and shares measurements of interest.

## II. Software Algorithm

An LDPC code may be expressed using an $M$x$N$ binary matrix $H$, where each row $M$ defines the variables in a parity-check set, and the number of columns $N$ matches the length of the block being decoded. This design implements the Min-Sum algorithm described by Chen et al. [4] for iterative decoding. The following terms are used in this work:

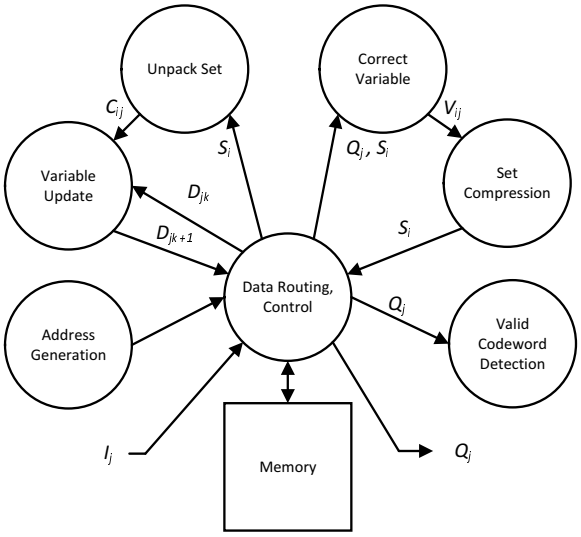


Fig. 1: High level software implementation of a Min-Sum LDPC decoder.

- $I_j$ Log-likelihood ratio of channel information for the $j$-th variable node, input to the decoder.
- $C_{ij}$ Message from check node $i$ to variable node $j$.
- $S_i$ Compacted Set containing messages $C_{ij}$ for a given $i$.
- $V_{ij}$ Message from variable node $j$ to check node $i$.
- $D_{jk}$ Partial sum of $k$ check node messages and channel information in variable node $j$.
- $Q_j$ Total sum of check node messages and channel information in variable node $j$.
- $L(i)$ Group of variable nodes connected to check node $i$.

Each decoding iteration is split into two major phases: Compact Set, which effectively collects and compresses the variable node messages $V_{ij}$ into set $S_i$, and Update Variable, which sums check node message $C_{ij}$ and $D_{jk}$ to generate $D_{jk+1}$, obtaining $Q_j$ when all summations are complete. Major data structures are maintained in shared memory. Fig. 1 shows the primary components and data connections for this software decoder.

### A. Parity Check Matrix

The parity check matrix is implemented by the memory access pattern. For each row of the matrix, a memory address generator will produce the group of addresses corresponding to the variables $L(i)$ in the parity set. Addresses may be

TABLE I: Parameters of implemented LDPC codes.

| Code | Rows | Columns | Row Weight | Column Weight |
|---|---|---|---|---|
| (4095,3367) | 378 | 4095 | 64 | 5 |
| (16129,15372) | 762 | 16129 | 127 | 6 |

generated either from an look-up table or, if the matrix has sufficient regularity, using a mathematical expression that is computed in software. This work implements designs for the two LDPC codes described in Table I.

### B. Compact Set

In the Min-Sum algorithm, the check node messages $C_{ij}$ for a given check node $i$ may only have one of two possible magnitudes, corresponding to the two minimum magnitudes of messages $V_{ij}$ from variable nodes $L(i)$ [4]. During Compact Set, for each $i$, $V_{ij}$ are gathered and processed to determine these two minimum magnitudes, along with the index $j$ of the first minimum, a compressed set of sign bits $N_j$ isolated from $V_{ij}$, and the total XOR of these sign bits $T$. The resulting compacted sets $S_i$ are stored in memory.

### C. Unpack Set

Compacted sets $S_i$ are unpacked to generate check node messages. For any given message $C_{ij}$, the magnitude is equal to the first minimum in $S_i$ unless $j$ matches the first minimum's index, in which case the magnitude is equal to the second minimum. The sign of $C_{ij}$ is equal to $T \oplus N_j$, the total sign bit corrected by the bit for variable $j$.

### D. Update Variable

The original channel information $I_j$ is combined with check node messages $C_{ij}$ to generate $D_{jk}$, the partial sums for each variable node. After the final summation, the $D_{jk}$ term is stored as $Q_j$. In this work, each set $S_i$ is processed in order, with the partial sums $D_{jk}$ read from memory, updated to form $D_{jk+1}$, and stored back into memory until their next corresponding set is processed. An alternative approach would be to process variable nodes $j$ in order and read corresponding sets $S_i$ as needed, but this results in greatly increased memory traffic for the selected LDPC codes.

### E. Correct Variable

Each $Q_j$ produced by Update Variable requires correction in order to obtain the variable node messages $V_{ij}$ to be passed to Compact Set on the following decoding iteration. In Correct Variable, the previous set $S_i$ is unpacked to obtain the message $C_{ij}$, as in Unpack Set above, and the operation $V_{ij} = Q_j - C_{ij}$ is performed to obtain the corrected messages.

### F. Valid Codeword Detection

Decoding is complete when the group of variables $Q_j$ in each parity group $L(i)$ satisfy parity, indicating a valid codeword has been found and may be output. This analysis may be performed in parallel with the Compact Set phase, sharing the input data stream of Correct Variable in order to overlap memory access. Detection of a valid codeword is communicated to all relevant nodes to trigger data output with a general reset in preparation for the next input $I_j$.

TABLE II: Memory utilization of data structures, with the number of 16-bit memory words required to store the structure, and the read/write activity during a single Compact Set (CS) or Update Variable (UV) iteration.

| Data, Code Length | Words | Reads CS | Writes CS | Reads UV | Writes UV |
|---|---|---|---|---|---|
| Input $I_j$, 4095 | 4095 | 0 | 0 | 4095 | 0 |
| Input $I_j$, 16129 | 8128 | 0 | 0 | 16129 | 0 |
| Variables $D_{jk}$, 4095 | 4095 | 24192 | 0 | 24192 | 24192 |
| Variables $D_{jk}$, 16129 | 8128 | 96774 | 0 | 80645 | 96774 |
| Sets $S_i$, 4095 | 2646 | 2646 | 2646 | 2646 | 0 |
| Sets $S_i$, 16129 | 7620 | 7620 | 7620 | 7620 | 0 |

## III. Software Implementation

### A. AsAP2 Platform

The AsAP2 computational platform [2] consists of an array of 164 programmable, 16-bit, RISC processors; three 16kB memories with two read/write ports each; and specialized accelerators for motion estimation, FFT, and Viterbi decoding. AsAP2 occupies 32.7 $\mathrm{mm}^2$ and was developed in 65 $\mathrm{nm}$ CMOS. A single processor may hold up to a 128 instruction program, contains a 128 word data memory, and communicates with other processors using a statically configured circuit network. A processor is limited to 2 input links and as many as 8 output links. Each processor contains its own clock oscillator which may run up to 1.2 GHz, and each processor may be powered by one of two voltage rails. Memory modules are limited to one random access per two cycles per port, and are accessed using 16-bit words with 8192 unique addresses.

### B. Memory Mapping, Data Routing

Three data structure are stored in the on-chip memory modules: the original input channel information $I_j$, the variable node data $D_{jk}$ or $Q_j$, and the compacted sets $S_i$. Table II describes the memory usage of each data structure, along with the read and write activity during the two major decoding phases. Data is routed using dedicated processors which perform the appropriate data stream joining and splitting functions.

The memory and data routing system for code length 4095 is shown in Fig. 2. Here, $I_j$ and $D_{jk}$ are stored in an interleaved fashion and split across two memory modules, allowing two read and two write ports to be active during Update Variable, and four read ports to be active during Compact Set. Fig. 3 shows this system for code length 16129. Here, $I_j$ and $D_{jk}$ are stored in separate memories, and are packed two per memory address due to memory constraints. Minor connections used for control signaling and valid codeword detection are present but omitted from the figures for clarity.

### C. Compact Set Lane

The Correct Variable and Compact Set operations are mapped to a scalable computation lane, shown in Fig. 4. This lane may be replicated and stacked vertically to provide

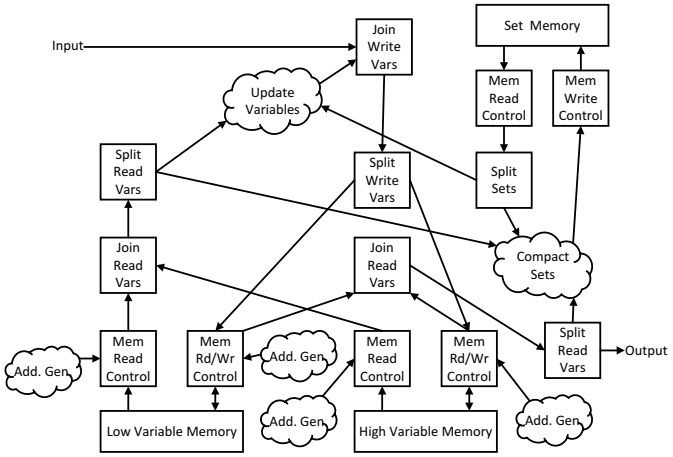


Fig. 2: Memory and data routing system for code length 4095.

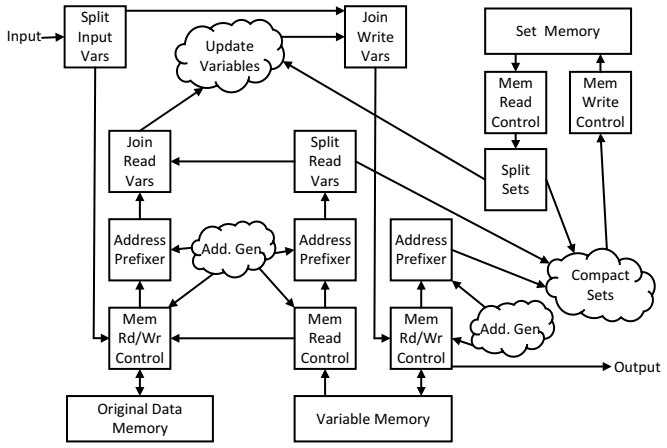


Fig. 3: Memory and data routing system for code length 16129.

parallel computing resources. Variable and set data is split across computation lanes upward, and the generated new Sets are joined and returned downward. In a final layout, some split and join cores may be omitted and replaced with direct data links if necessary. For the 4095 code, each lane is capable of processing two sets simultaneously.

For the 16129 code, additional overhead is required due to memory packing. Variable data read out of memory is packed with two variables per word, and must be unpacked by selecting the correct upper or lower Byte based on the variable's Byte address, which is included in the data stream. The unpacking core also provides additional buffering, which is needed to capture the data bursts which otherwise are too large for the standard input buffer inside each processor and will cause pauses in the variable data stream. Finally, since valid codeword detection must use the unpacked variable data, an additional core is included within the lane for performing this detection after unpacking. For the 4095 code, valid detection cores may access the variable stream directly, and are placed elsewhere.

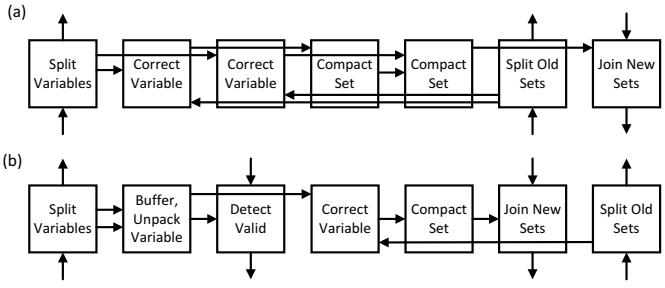


Fig. 4: Core mapping for a Compact Set computation lane, for code lengths (a) 4095 and (b) 16129. Lanes are replicated vertically to increase parallel computation.

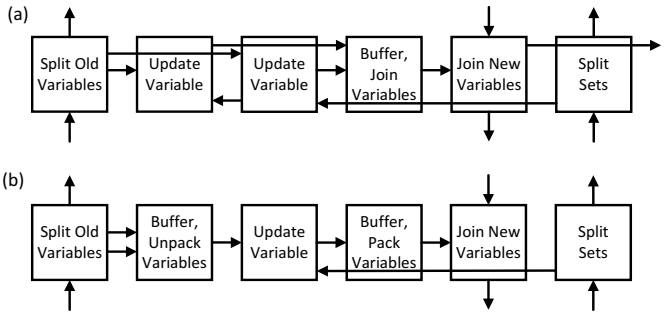


Fig. 5: Core mapping for an Update Variable computation lane, for code lengths (a) 4095 and (b) 16129. Lanes are replicated vertically to increase parallel computation.

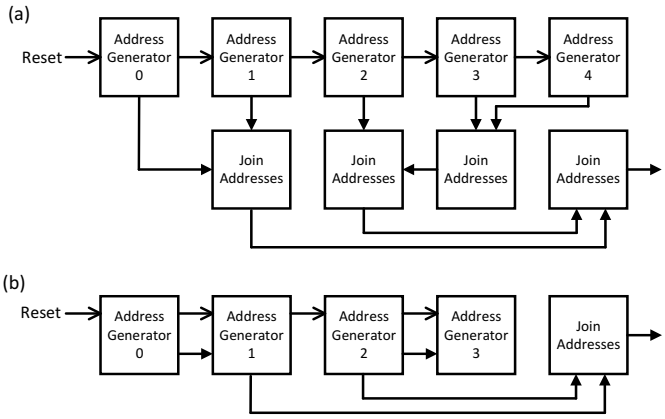


Fig. 6: Core mapping for an address generation block, for code lengths (a) 4095 and (b) 16129. These blocks are used to implement the parity check matrix $H$.

### *D. Update Variable Lane*

Similar to Correct Variable lane, the Update Variable lane is scalable and implements the Unpack Set and Update Variable operations, shown in Fig. 5. These two operations are performed inside of a single core, named Update Variable. For the 4095 code, each lane is capable of processing two sets simultaneously.

For the 16129 code, memory packing again imposes overhead. In this case, however, the unused variable in a packed pair must be preserved and rejoined with the updated variable before being written back to memory, to avoid data loss. If

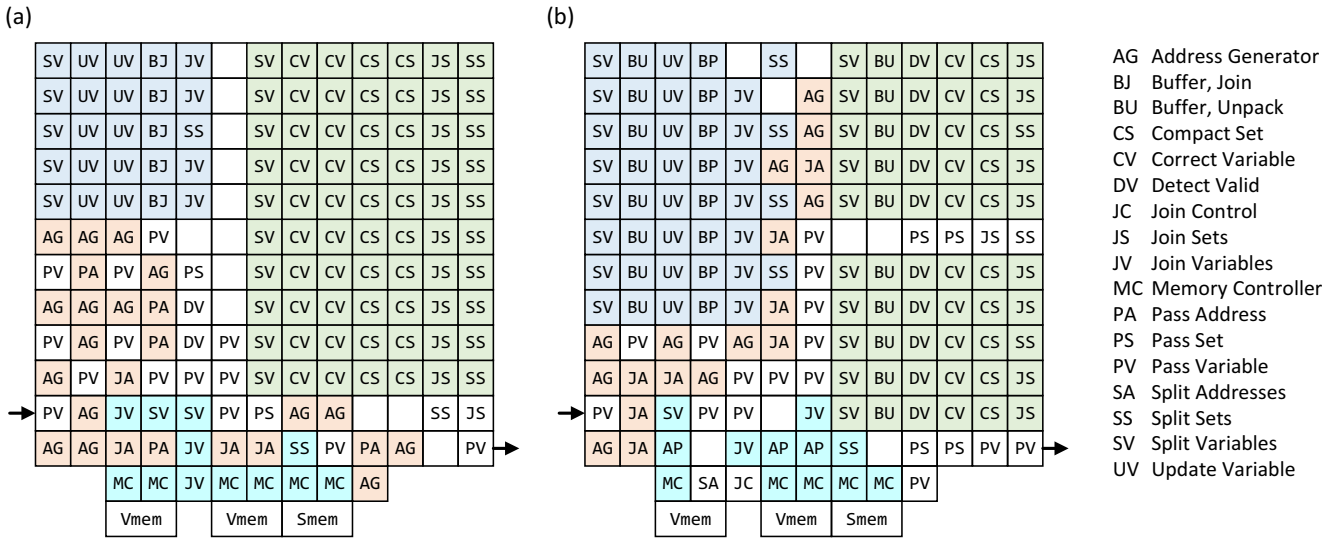


Fig. 7: Overall application mapping to AsAP2, for code lengths (a) 4095 and (b) 16129. Update Variable lanes are replicated in the upper left (shaded blue), Compact Set lanes are replicated in the upper right (shaded green), address generators (shaded orange) and data routing cores (shaded cyan) are clustered around the memory modules on the bottom.

this is not done here, the repacking must be done in a more expensive fashion at the memory interface core.

### *E. Address Generation*

Variable addresses are generated in software using a mathematical function which expresses the implemented parity check matrices. This function is capable of fitting within a single core's instruction memory, but has been partitioned across multiple cores for increased throughput, as shown in Fig. 6. Adaptation of this design to other LDPC codes may be done by changing the function in the address generator cores. The reset signal shown is sent when a valid codeword is detected.

## IV. Results

The computation blocks described in Section III were mapped to the 164 processor array in AsAP2. Compact Set and Update Variable compute lanes were each replicated until they were capable of processing data as quickly as it is read from memory during each phase of the application, and address generators were expanded to satisfy the address consumption rate of the memories. Due to a limitation of the AsAP2 architecture, cores sending data across more than two processors are limited to less than their normal maximum frequency. This is accounted for in the mappings, with additional Pass cores inserted along long, high-rate data links; Pass cores simply pass their input to their output. The final mappings are shown in Fig. 7, where Update Variable lanes are in the upper left, Compact Set lanes are in the upper right, and other cores are intermixed and generally clustered around the memories at the bottom of the array.

The instruction memory utilization of the cores is shown in Fig. 8, where cores are grouped into categories. The Correct Variable and Variable Memory Control cores show the largest

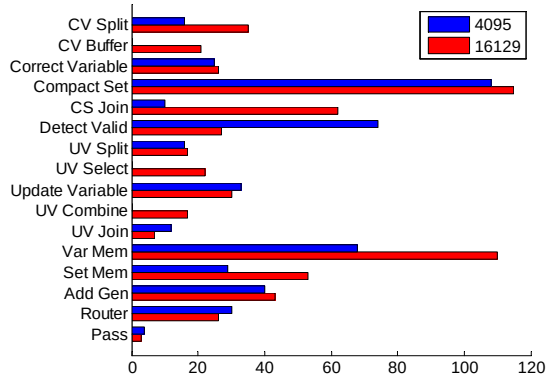


Fig. 8: Instructions used, given as the highest for any core within a category, for code lengths 4095 and 16129. Cores are limited to 128 instructions.

instruction counts due to loop unrolling. A core may hold a maximum of 128 instructions, but this limit was not found to be restrictive in this application.

Energy usage of the final mapping is optimized through an iterative profiling technique. Starting with all cores set to their maximum frequency, the frequency of each individual core is lowered until the overall application throughput is significantly reduced, with this point being recorded as the estimated minimum required frequency of the core. After all minimum frequencies are found, the second voltage rail in AsAP2 is set to the estimated minimum energy point, where lower frequency cores operate at a lower voltage for energy savings. Fig. 9 shows the average energy reduction for cores in each category. In some situations, energy usage may increase if a low frequency core communicates with a higher frequency core, leading to additional empty cycles in the receiving core as it waits for data to arrive. Overall, this method reduces energy usage by 9.3% and 12.1% with a

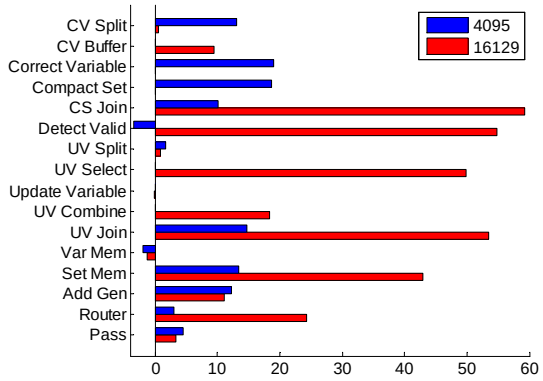


Fig. 9: Percentage by which energy usage is reduced by optimization, given as the average for cores within a category, for code lengths 4095 and 16129. Reductions are achieved through optimization of the frequencies and voltage rail selection of each individual core.

reduction in throughput of 0.56% and 0.44% for code lengths 4095 and 16129 respectively.

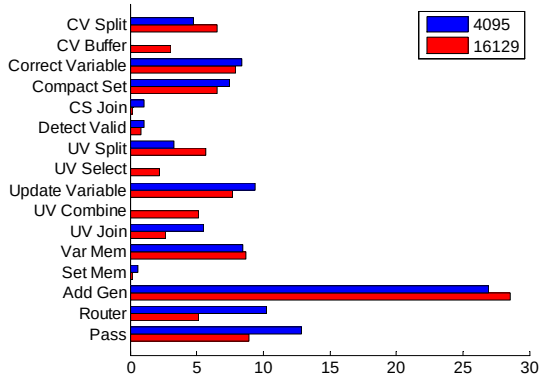


Fig. 10: Percent of the total application energy used by cores in each category, for code lengths 4095 and 16129.

Fig. 10 shows the overall energy usage of cores in each category, after optimization. The largest contributors are the address generators, which experience high activity during both phases of the application and are replicated multiple times to satisfy the address consumption rate of the memory controllers. The second largest contributors are Pass cores, which are typically placed on high rate data links and see high activity as a result.

The performance of this design is compared to two other software LDPC decoder implementations, as shown in Table III. First is a multi-threaded decoder written in C++ using the algorithm described in this work, and running on an Intel i7-3770k processor. Second is a GPU decoder presented by Li et al. [5] and running on an Nvidia GTX 580. Designs are scaled to 22 nm using the scaling equations presented by Stillmaker et al. [6], as well as being presented unscaled. Power for the i7 and GTX is assumed to be half of their rated Thermal Design Power. The primary metrics of interest are throughput per area and bits decoded per unit of energy, in both of which AsAP2 performs well.

TABLE III: Performance of this work compared to a C++ implementation and a GPU implementation. Primary metrics are throughput per area and bits decoded per unit of energy. Results are scaled to 22nm for comparison. Throughput is for 4 full decoding iterations and a partial 5th iteration, or 5 full iterations in the GPU implementation. AsAP2 performance is given with and without voltage optimization.

| Platform | Block Size | Tech. (nm) | Thr. (Mbps) | Thr./Area (Mbps/mm$^2$) | Bits/Energy (b/µJ) |
|---|---|---|---|---|---|
| i7-3770k | 4095 | 22 | 23.9 | 0.150 | 0.62 |
| i7-3770k | 16129 | 22 | 25.0 | 0.156 | 0.65 |
| GTX 580 [5] | 2304 | 40 | 710.0 | 1.365 | 5.82 |
| GTX 580 [5] | 2304 | 22 | 970.7 | 5.431 | 23.40 |
| AsAP2 | 4095 | 65 | 21.4 | 0.655 | 7.06 |
| AsAP2 | 16129 | 65 | 13.5 | 0.413 | 4.72 |
| AsAP2, Opt. | 4095 | 65 | 21.3 | 0.651 | 7.66 |
| AsAP2, Opt. | 16129 | 65 | 13.4 | 0.412 | 5.31 |
| AsAP2 | 4095 | 22 | 85.3 | 11.735 | 45.71 |
| AsAP2 | 16129 | 22 | 53.8 | 7.411 | 30.52 |
| AsAP2, Opt. | 4095 | 22 | 84.8 | 11.670 | 50.40 |
| AsAP2, Opt. | 16129 | 22 | 53.6 | 7.379 | 34.74 |

## V. Conclusion

This work has presented the design and implementation of two software LDPC decoders on the AsAP2 many-core platform, supporting code lengths of 4095 and 16129. The implementations achieve a high throughput per area while maintaining a low energy per decoded bit. Up to 21.3 Mbps throughput is achieved for 4095 length blocks, decoding 7.66 bits per µJ. For 16129 length blocks, throughput and energy efficiency are 13.4 Mbps and 5.31 bits/µJ respectively. These designs may be adapted to other LDPC codes by selective modification of the address generation units, with the 4095 design supporting codes up to length 8192, and the 16129 design supporting codes up to length 16384.

## VI. Acknowledgements

The authors gratefully acknowledge support from C2S2 Grant 2047.002.014, NSF Grant 1018972 and 0903549 and CAREER Award 0546907, SRC GRC Grant 1598, 1971, and 2321 and CSR Grant 1659.

## References

[1] R. Gallager, "Low-density parity-check codes," *Information Theory, IRE Transactions on*, vol. 8, no. 1, pp. 21–28, January 1962.

[2] D. Truong *et al.*, "A 167-processor computational platform in 65 nm cmos," *Solid-State Circuits, IEEE Journal of*, vol. 44, no. 4, pp. 1130–1144, April 2009.

[3] J. Bisasky, D. Chandler, and T. Mohsenin, "A many-core platform implemented for multi-channel seizure detection," in *Circuits and Systems (ISCAS), 2012 IEEE International Symposium on*, May 2012, pp. 564–567.

[4] J. Chen *et al.*, "Reduced-complexity decoding of ldpc codes," *Communications, IEEE Transactions on*, vol. 53, no. 8, pp. 1288–1299, Aug 2005.

[5] R. Li *et al.*, "A multi-standard efficient column-layered ldpc decoder for software defined radio on gpus," in *Signal Processing Advances in Wireless Communications (SPAWC), 2013 IEEE 14th Workshop on*, June 2013, pp. 724–728.

[6] A. Stillmaker, Z. Xiao, and B. Baas, "Toward more accurate scaling estimates of cmos circuits from 180 nm to 22 nm," VLSI Computation Lab, ECE Department, University of California, Davis, Tech. Rep. ECE-VCL-2011-4, Dec. 2011, http://www.ece.ucdavis.edu/cerl/techreports/2011-4/.