Design and Programming of the KiloCore Processor Arrays

By

Brent Vince Bohnenstiehl
Dissertation

Submitted in partial satisfaction of the requirements for the degree of

Doctor of Philosophy

in

Electrical and Computer Engineering

in the

Office of Graduate Studies

of the

University of California

Davis

Approved:

Professor Bevan Baas, Chair

Professor Soheil Ghiasi

Professor Venkatesh Akella

Committee in Charge

2020

Abstract

Design and Programming of the KiloCore Processor Arrays

Modern semiconductor fabrication technologies now enable the construction of integrated circuits which contain over 1000 processors on a single chip [1]. However, for such systems to effectively compute workloads, new architectures are needed for the processors, the inter-processor interconnect, circuits that interact with larger memories, and the applications they execute [2–4].

This work explores the characteristics of many-core arrays, and utilizes gained insights to advance the state of the art through a new architecture designed to efficiently scale to thousands of processors per chip, along with software development tools to aid in effectively programming such arrays.

A detailed exploration is made of prior many-core architecture work to identify areas that might be significantly improved. Possible benefits are found in the instruction set selection, pipeline design, network communication between processors, voltage control logic, and other areas. To name a few findings: inclusion of unsigned support speeds some operations up by as much as 15x, profile-guided static branch prediction raises the correct prediction rate from 27% to 96% in sampled applications, fast oscillator halting reduces active-clock stall cycles by 33%, and voltage dithering improves DVFS energy savings by 16%.

A 1,000-processor array named KiloCore is presented. Fabricated in 32 nm PD-SOI CMOS technology and occupying 64 mm^2, this newly designed architecture implements lessons learned from prior work along with other innovations. KiloCore processors may operate up to 1.78 GHz at 1.1 V, and down to 115 MHz at 560 mV where an operation dissipates only 5.8 pJ. Several applications are implemented on KiloCore and their characteristics and performance are discussed. Across these applications and when scaled to the same fabrication technology, KiloCore at 0.9 V has geometric mean improvements of 3.1x higher throughput per area and 16.7x higher energy efficiency compared to published results on standard CPU and GPU architectures. Comparing to just-CPUs and just-GPUs, Kilocore achieves 68.9x and 72.0x higher throughput per area per Watt respectively.

A followup 697-processor array named KiloCore2 is presented. Fabricated with the same technology and chip area as KiloCore, KiloCore2 is designed to achieve a 63% higher throughput per processor than its predecessor, supports three voltage domains for optimizing per-processor

energy efficiency, implements a new low-area packet routing network that is specifically designed for the needs of a many-core system, and adds FFT and Viterbi accelerators along with a selection of high speed processors for speeding up serial code. Pending final measurements, KiloCore2 is designed to reach 2.0 tera-operations per second at 1.1 V, with standard processors reaching 2.9 GHz and fast processors reaching over 5.0 GHz operation.

Programming and analysis tools for many-core arrays are presented. High speed simulators, written in C++, are over 50,000 times faster than Verilog RTL simulation, and contain a suite of features to aid in application development and debugging. The KiloCore compiler generates optimized assembly from user supplied kernels written in C++, Python, or potentially other languages. Leveraging the LLVM infrastructure to act as a front end, this compiler focuses on the back end operations needed to lower LLVM IR code into the format needed for stackless, 16-bit, direct-memory-access processors with strict memory limitations, as well as optimize the code to be comparable to optimized hand-written assembly. Supplementing these tools is a Project Manager, which allows users to write simple Python scripts to define a collection of tasks, replicate and map them to a target many-core array, perform array-wide optimizations, and to conveniently compile and run their applications.

Acknowledgments

I give particular thanks to my advisor, Dr. Bevan Baas. In addition to providing funding and a position in the VLSI Computation Laboratory (VCL) at UC Davis, his prior work on many-core arrays was a critical inspiration for my own research. The KiloCore and KiloCore2 test chips would not have been possible without Dr. Baas' connections and efforts in organizing the fabrications.

While the work presented in this dissertation represents my own original efforts, some parts touch on the work of others. I thank these current and prior members of the VCL group for their contributions:

- Aaron Stillmaker for managing the physical design toolchain for the KiloCore chips.
- Timothy Andreas for daughterboard design of both chips, lab equipment setup, and oscillator refinements in KiloCore2.
- Mark Hildebrand for writing a task-to-core mapping tool for many-core arrays.
- Jon Pimentel for assisting with the physical designs of both chips.
- Bin Liu for the oscillator design in KiloCore, reused in KiloCore2, and for assisting in the physical design of KiloCore.
- Anh Tran for adapting his packet router design for use in the first KiloCore.
- Emmanuel Adeagbo for assisting with the physical designs of both chips, and oscillator refinements in KiloCore2.
- Prior members of the VCL group for their earlier work on AsAP2 and its applications, which forms the background for this research, and for the FFT and Viterbi accelerators which were utilized in KiloCore2.

I thank Jeremy Rodgers for designing the chip package for KiloCore2.

I thank Dr. Venkatesh Akella, Dr. Soheil Ghiasi, and the other professors in the Electrical and Computer Engineering department at UC Davis.

I thank my family members, on two legs or four, for their support and companionship through the years of this research.

This work was supported by DoD and ARL/ARO Grant W911NF-13-1-0090; NSF Grants 0903549, 1018972, 1321163, and CAREER Award 0546907; SRC GRC Grants 1971 and 2321, and CSR Grant 1659; and C2S2 Grant 2047.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Parallel processing offers well known benefits in performance and efficiency, with many modern chip designs focusing on integrating an increasing numbers of processors on a single die instead of increasing the complexity of a smaller number of processors [2–6]. Many current and future computing applications, ranging from embedded internet-of-things devices to cloud datacenters, are placing increased emphasis on hardware solutions that provide high energy efficiency alongside high performance [7].

Semiconductor fabrication technologies continue to provide increasing levels of integration [8], offering opportunities for new architecture designs. However, increasing fabrication costs continue to motivate the development of programmable and/or reconfigurable architectures which can address the needs of a range of applications in varying computing domains. With the looming end of Moore's Law, such non-standard architectures and programming techniques enable compute performance to continue to scale. Figure 1.1 shows the trend of an increasing number of processors per die, where each processor is capable of independent program execution.

The highest energy efficiency and throughput-per-area is found when processors are a direct match for the operational and memory requirements of their programs. This leads to an architectural design problem: as processor size is reduced for efficiency when running small applications, effectiveness or capability to run large applications is lost. Multiple-processor arrays offer a solution: applications can be separated into groups of tasks better suited for fine-grain processors, with the full array working together to support an entire application [9].

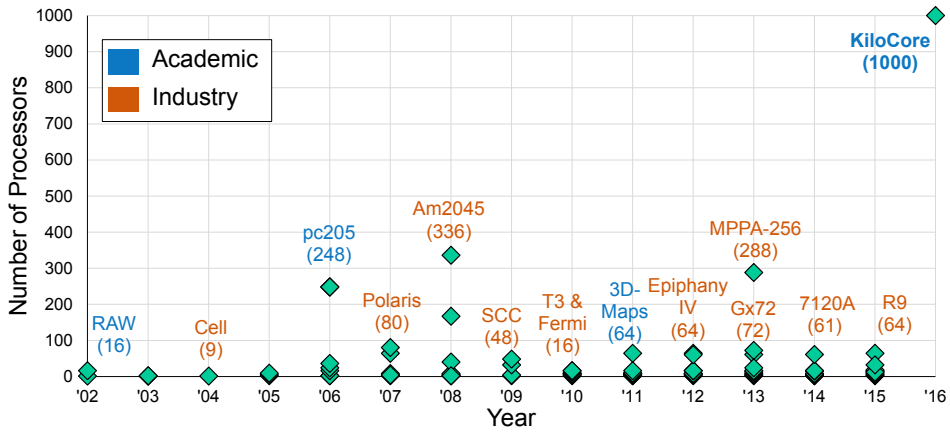


Figure 1.1: Number of processors on a single die vs. year of publication, up to the publication of KiloCore in 2016. Each processor is capable of independent program execution.

Compared to traditional CPU architectures, many-core arrays are poorly suited to large, general applications, but excel as reprogrammable accelerators for streaming applications. In streaming compute, a short algorithm, typically represented by hundreds to thousands of lines of code, operates on a series of data blocks, where blocks may arrive over time (eg. processing wireless signals) or be read from a larger database (eg. encrypting data). The accelerator is programmed with one or more such algorithms, and enables significantly faster or lower energy processing of the data.

This fine-grain approach has been used to develop processor arrays consisting of tens to hundreds of processors [4, 5, 10, 11]. However, the industry-developed architectures [10, 11] are opaque with respect to details of their microarchitecture, performance in applications, and programming approach. The university-developed architectures [4, 5] offer transparency, but are lacking in hardware refinement and programming tools, and do not best represent what many-core architectures are capable of.

To address these issues, the research presented here develops a next-generation many-core architecture. This refined design better represents the performance advantages possible with many-core arrays. Actual performance in silicon is measured and shared publicly, along with the details of the architecture. Further, to address difficulties in programming such arrays, a suite of software tools are developed to aid in designing applications consisting of hundreds to thousands or more tasks.

An ambitious goal is set for this new architecture: to achieve over a 50x improvement in throughput per area per energy when compared to traditional CPUs and GPUs running

comparable, externally-developed applications. For fairness, all compared chips are scaled to the same fabrication technology, and unused die area is omitted (eg. the on-die graphics accelerator in some modern CPU chips).

Designed to scale to thousands of processors per chip, this new architecture has been named KiloCore [9, 12–15]. KiloCore addresses the aforementioned factors with a massively-parallel computing platform that is energy efficient for a wide variety of workloads, capable of very high performance, easily scalable to higher processor counts, and suitable for a range of applications and critical kernels either alone or acting as a co-processor in a heterogeneous system. Fabricated in 32 nm PDSOI CMOS, KiloCore exceeds the aforementioned goal by achieving 70x higher throughput per area per Watt than traditional CPUs and GPUs, and ushers in the era of 1,000+ independent processors per chip.

1.2 Related Work

Multiple-processor architectures, where each core runs an independent program, can be loosely categorized into three varieties, as follows. These are summarized in Table 1.1.

1. Coarse-grain multi-core arrays implement processor cores with approximately 0.2 to 1 MB or more memory per core [16, 17]. Such processors operate largely independently on general purpose complex workloads, with strong support for existing applications and requiring little programmer redesign effort to achieve high core utilization. Common features include multi-threading, multi-issue, out-of-order execution, and single-instruction-multiple-data operations.

2. Medium-grain arrays provide 10 KB to 100 KB of memory per core [6, 18]. Increased emphasis is placed on work sharing between cores to realize the architecture's potential performance, where a single coarse-grain thread may be split into a handful of subtasks.

3. Fine-grain many-core arrays provide around 1 KB to 5 KB of memory per core [4, 10, 11]. Implementing complex applications requires a high degree of code partitioning and shared work between cores, with a single coarse-grain thread being split into tens or hundreds of subtasks.

Fine-grain architectures offer the highest potential energy and throughput efficiency, due to the low energy per operation for each core and the high number of cores which fit into a

Table 1.1: Selection of chips showing the range of processor sizes in many-core arrays, categorized as coarse-grain (200 kB or more memory per processor) down to fine-grain (less than 5 kB per processor).

Granularity	Creator	Chip	Processors	Memory per Proc	Year
Coarse	Intel	Single-chip Cloud Comp.	48	276 kB	2010
Coarse	Intel	Phi 5110P	60	544 kB	2012
Coarse	Knupath	Hermosa	256	280 kB	2016
Medium	Tilera	TILE64	64	80 kB	2007
Medium	Kalray	MPPA-256	256	144 kB	2012
Medium	Adapteva	Epiphany V	1024	64 kB	2016
Fine	Picochip	PC101	430	768 B or 8 kB or 32 kB	2003
Fine	Ambric	Am2045	336	288 B or 5 kB	2008
Fine	UCDavis	AsAP2	164	848 B	2008

given area. Such architectures are the focus of this research. Notable fine-grain architectures are described below.

The Picochip PC101 [10] contains 430 16-bit, VLIW processor cores operating at 160 MHz. Typical cores have 768 B of memory. The cores are heterogeneous: 240 are basic cores, 120 cores add multiply-accumulate (MAC) support, 68 cores add 8 kB of memory, and 2 control cores add 32 kB of memory. Subsequent versions reduced the number of cores per chip, replacing the basic cores with fewer MAC cores. A time multiplexed synchronous mesh network is used for communication, with data transferred horizontally along rows and vertically every four columns, and cores being required to wait for their scheduled slot before performing a transfer.

The Ambric Am2045 [11] contains 336 32-bit processor cores operating at 300 MHz. The cores are heterogeneous: 168 cores have 288 B of memory, a limited instruction set, and are use primarily for simple tasks such as network data routing; the other 168 cores have 5 kB of memory and the full instruction set. Two of each type of core, four total, form a compute unit. Cores in a compute unit may communicate directly with each other, while the network connects compute units to neighbors and allows them to form longer distance connections through a switching mesh using handshake synchronization.

The UC Davis Asynchronous Array of Processors 2 (AsAP2) [4] contains 164 16-bit homogeneous processor cores operating up to 1.2 GHz. Each processor has 1 kB of local memory, with an additional 48 kB of shared memory located at the bottom of the chip. A source-synchronous, two-layer, circuit-switched mesh network is used for all communication.

Of these fine-grain arrays, AsAP2 is the only university chip and has published a much greater level of detailed information on its architecture, energy efficiency, and application performance. As such, it is the primary starting point for this research, and is described in further detail below.

1.2.1 AsAP2 Platform

The AsAP2 computational platform [4] supplements its 164 processors with three 16 kB memories having two read/write ports each, and with specialized accelerators for motion estimation, FFT, and Viterbi decoding. AsAP2 occupies 32.7 mm^2 and was developed in 65 nm CMOS. Figure 1.2 shows the high level layout of the array.

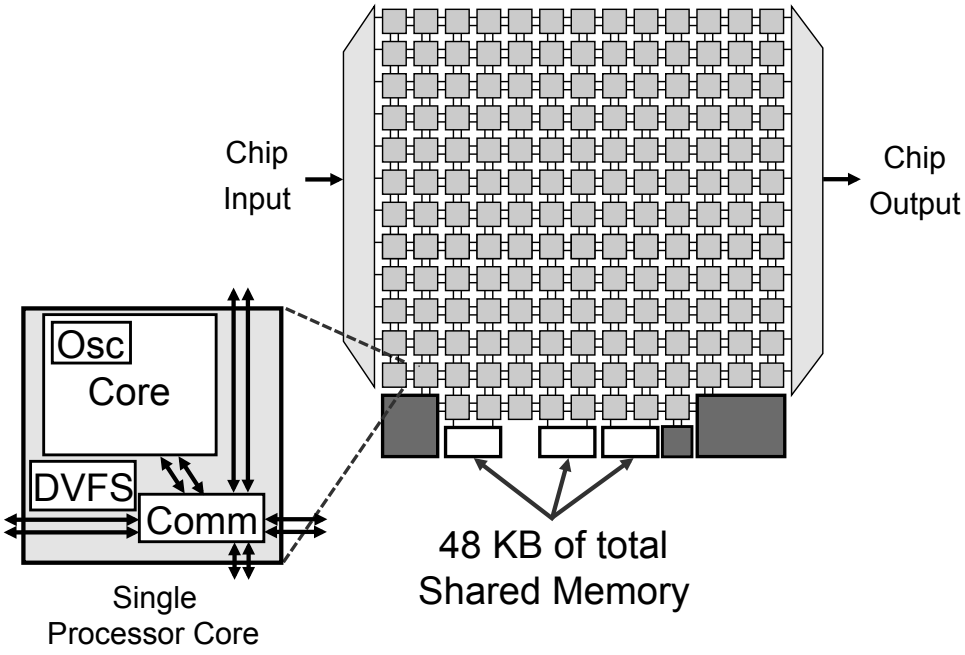


Figure 1.2: High level block diagram of the AsAP2 processor array [4].

A single 16-bit 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 and 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.

Figure 1.3 shows the six-stage pipeline diagram of a processor. Branch logic (not shown) is placed alongside instruction decode. Figure 1.4 shows the block diagram of the FFT accelerator.

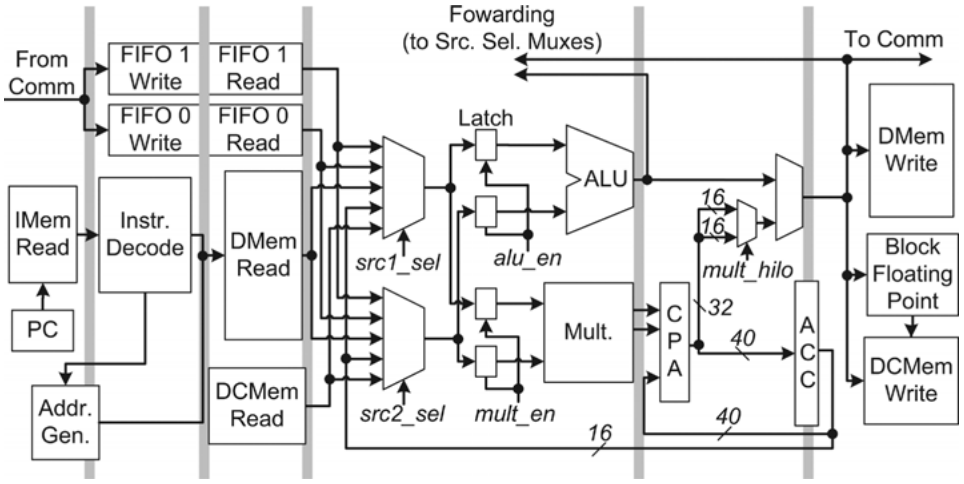


Figure 1.3: Block diagram of the six-stage pipeline of a single AsAP2 processor [4].

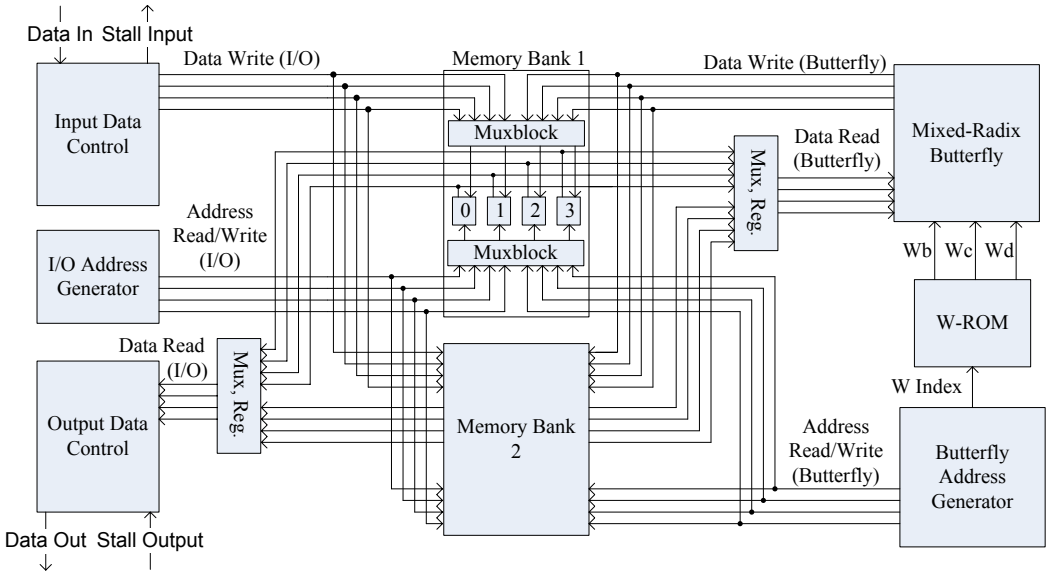


Figure 1.4: Block diagram of the FFT accelerator in AsAP2 [19], reused with modifications in KiloCore2.

1.2.2 AsAP2 Applications

A number of applications were developed to run on the AsAP2 platform, several of which are analyzed or adapted as part of this work. A brief overview of several such applications follows.

An Advanced Encryption Standard (AES) engine operates on 128-bit keys, and uses up to 137 cores [20], with several variations of lower core count. Figure 1.5 displays the task connections,

where a block of 14 tasks acts as a template that is replicated to nine instances, with final data processing happening in a block of 11 tasks.

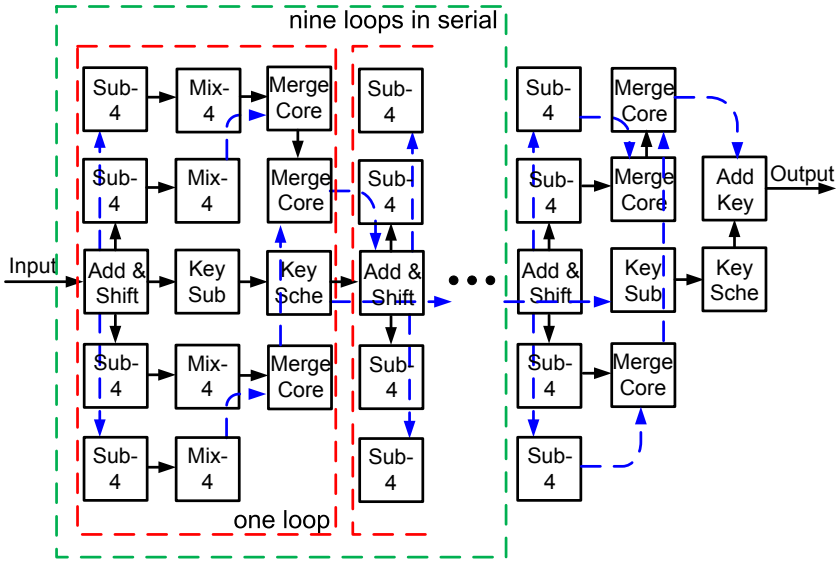


Figure 1.5: Tasks used in a 137-processor AES engine designed for AsAP2 [20].

The first phase of an "external" record sort is implemented for a variable number of processors [21]. Records are processed into sorted blocks in support of the second merging phase of the external sort. The primary version processes 100-byte records containing 10-byte sorting keys, while a secondary version operates on 4-Byte records containing 2-byte keys. In the "Snakesort" layout, processors are connected in a single, linear series, from array input to output. In the "Rowsort" layout, records are distributed across several parallel "Snakes", and merge-sorted together at the array output. Figure 1.6 shows the tasks involved in these sorting algorithms, after having been mapped to the full AsAP2 array.

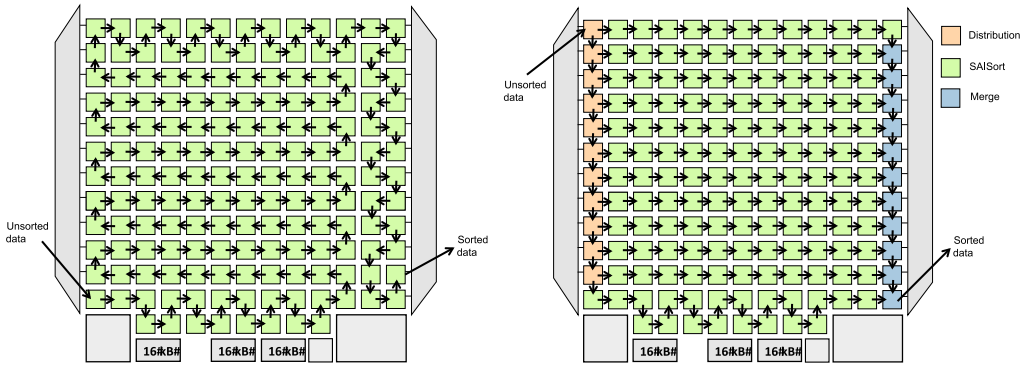


Figure 1.6: Tasks and mapping for Snakesort (left) and Rowsort (right) for AsAP2 [21].

An 802.11a baseband receiver [22] uses 23 processors, and utilizes the AsAP2 FFT and Viterbi accelerators. Figure 1.7 displays the task connections and accelerator usage.

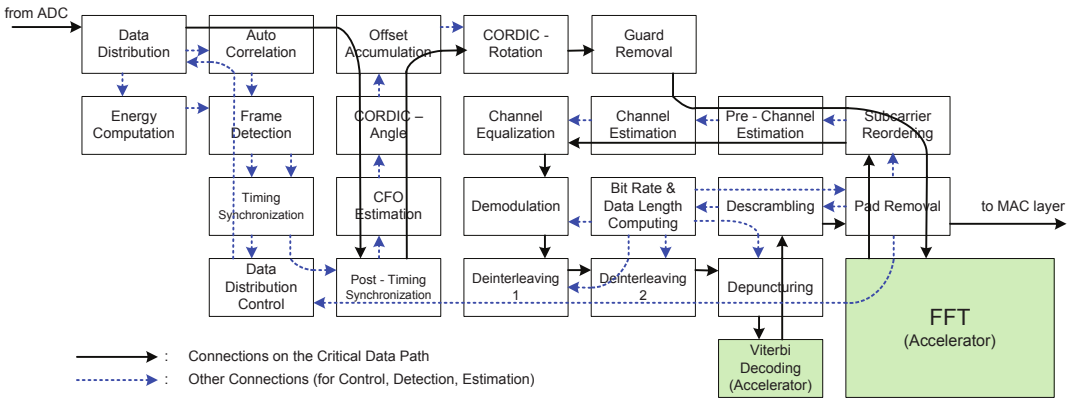


Figure 1.7: Tasks used in a 23-processor 802.11a baseband receiver for AsAP2 [22].

An H.264/AVC encoder is implemented using 147 processors [23], and utilizes the AsAP2 motion estimation accelerator. Figure 1.8 displays high level block diagram of the encoder, including the accelerator usage.

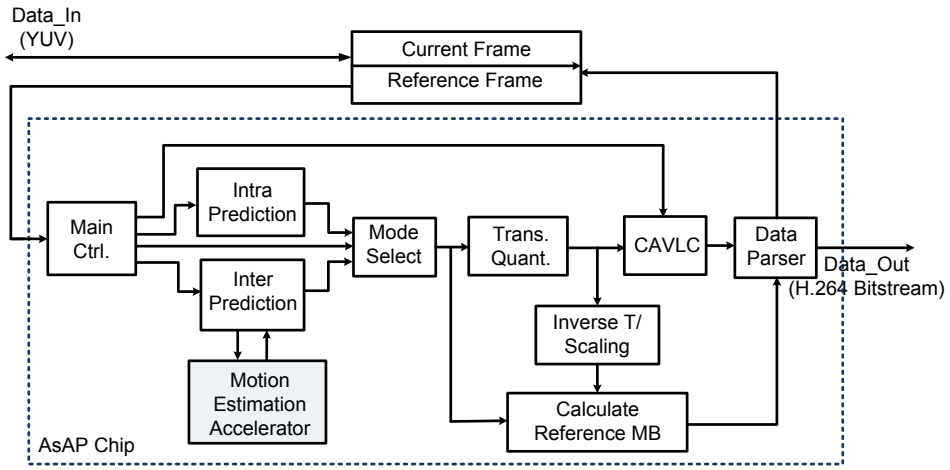


Figure 1.8: Block diagram of the 147-processor H.264/AVC encoder for AsAP2 [23].

1.3 Dissertation Organization

The remainder of this document is organized as follows. Chapter 2 introduces a Low Density Parity Check decoder application, to build an understanding of designing software to utilize many-core arrays. Chapter 3 shares insights derived from the prior AsAP2 work which led to the development of KiloCore. Chapter 4 details the KiloCore architecture and fabricated chip,

including measurement results. Chapter 5 details the followup KiloCore II architecture and chip, focusing on differences from KiloCore. Chapter 6 presents the software tools developed to aid in design and analysis of many-core applications as well as architectural exploration. Chapter 7 summarizes the results of the research presented here.

Chapter 2

A Low Density Parity Check Decoder for a Many-Core Array

Critical to understanding the design of many-core arrays is understanding how their applications are developed. This chapter explores the design of such an application in detail.

2.1 Introduction

Low Density Parity Check (LDPC) codes, introduced in 1962 by Gallager [24], 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 involves 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.

Fine-grain, many-core architectures offer high performance at low power for DSP applications, and are a promising platform for software-based LDPC decoding.

A software decoding algorithm is described in section 2.2. Section 2.3 describes two implementations of the algorithm on AsAP2 for sample LDPC codes of different lengths. Section 2.4 presents the final designs and shares measurements of interest.

2.2 Software Algorithm

An LDPC code may be expressed using an MxN binary matrix H, where each of the rows (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 utilizes the Min-Sum algorithm described by Chen et al. [25] for iterative decoding. The following terms are used in this work:

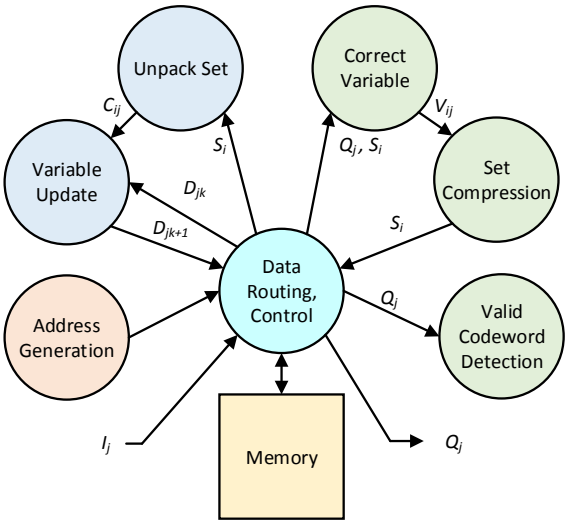


Figure 2.1: High level software implementation of a Min-Sum LDPC decoder.

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 capable of generating 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 information from the variable node messages V_{ij} into set S_i, and Update Variable, which sums check node messages 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. Figure 2.1 shows the primary components and data connections for this software decoder.

2.2.1 Parity Check Matrix

The parity check matrix is implemented through 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 generated either using a look-up table or, if

Table 2.1: Parameters of implemented LDPC codes.

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

the matrix has sufficient regularity, by computing them in software. This work uses the latter method to implement the two LDPC codes described in Table 2.1.

2.2.2 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 $L(i)$ [25]. 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.

2.2.3 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.

2.2.4 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.

2.2.5 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

Table 2.2: 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	20097	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

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.

2.2.6 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.

2.3 Software Implementation

2.3.1 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 2.2 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 Figure 2.2. 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. Figure 2.3 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

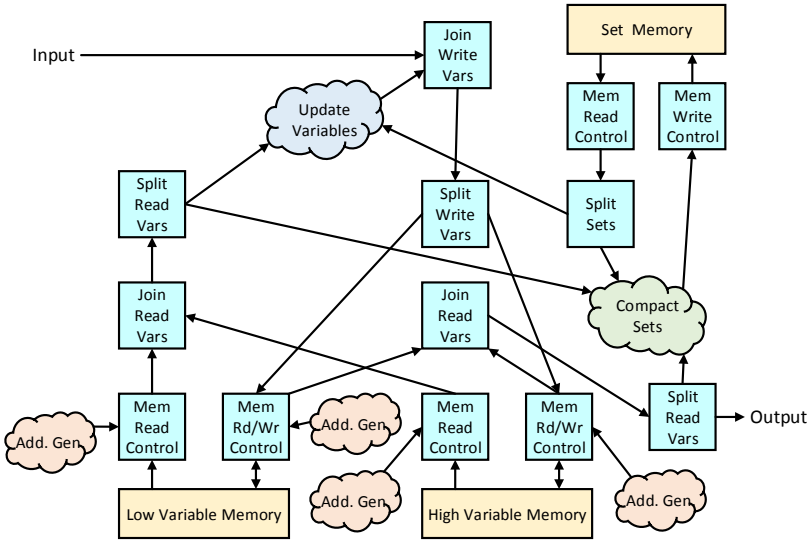


Figure 2.2: Memory and data routing system for code length 4095.

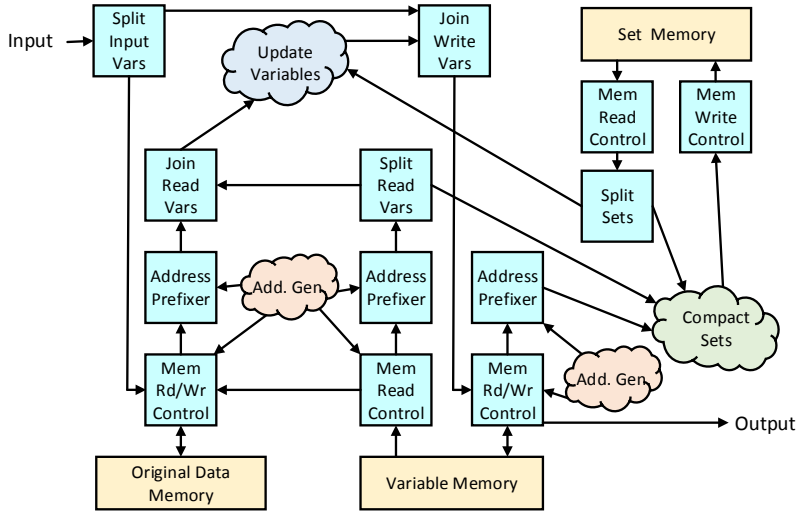


Figure 2.3: Memory and data routing system for code length 16129.

constraints. Minor connections used for control signaling and valid codeword detection are present but omitted from the figures for clarity.

2.3.2 Compact Set Lane

The Correct Variable and Compact Set operations are mapped to a scalable computation lane, shown in Figure 2.4. This lane may be replicated and stacked vertically to provide 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

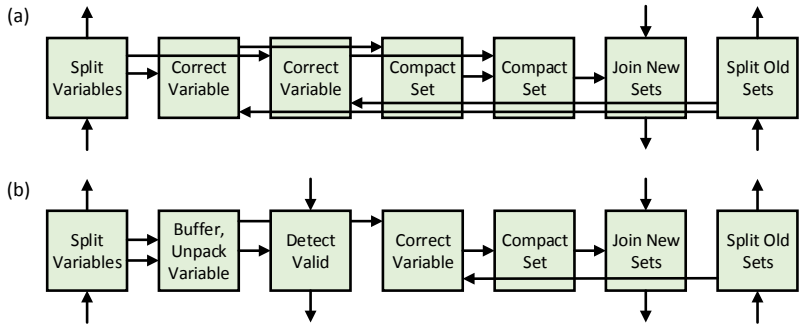


Figure 2.4: Core mapping for a Compact Set computation lane, for code lengths (a) 4095 and (b) 16129. Lanes are replicated vertically to increase parallel computation.

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 aids in capturing data bursts which are larger than a processor's input buffer and would otherwise cause back-up in the variable splitting cores. 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.

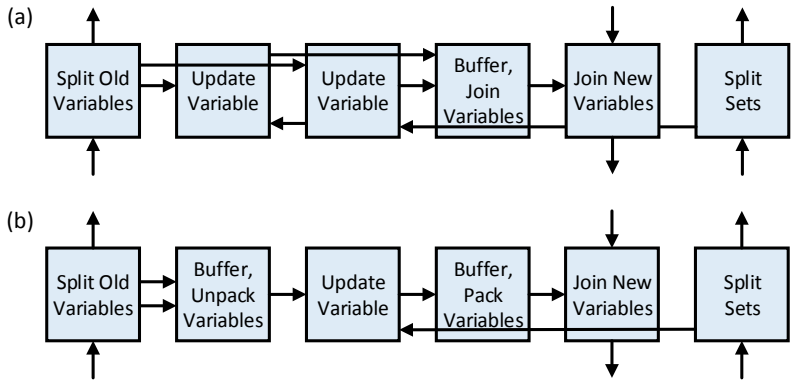


Figure 2.5: Core mapping for an Update Variable computation lane, for code lengths (a) 4095 and (b) 16129. Lanes are replicated vertically to increase parallel computation.

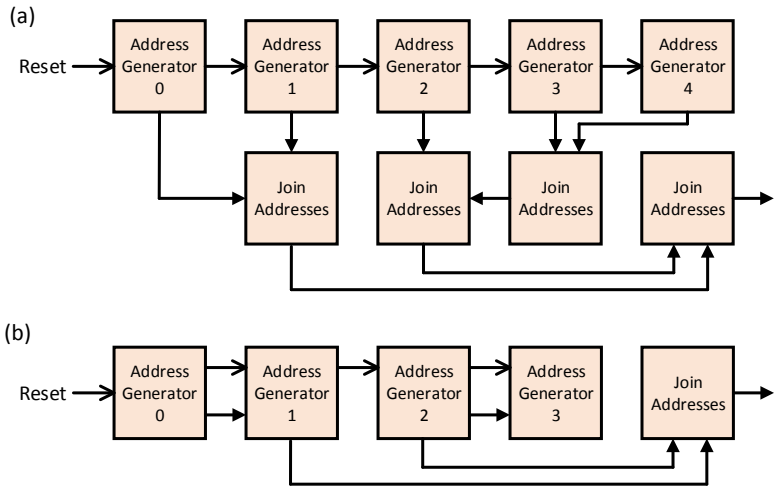


Figure 2.6: Core mapping for an address generation block, for code lengths (a) 4095 and (b) 16129. These blocks are used to implement the parity check matrix H.

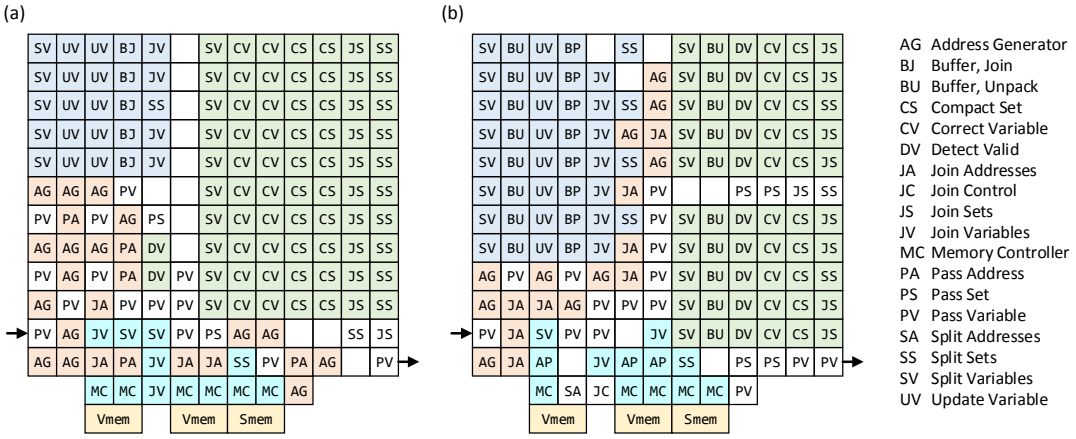


Figure 2.7: Overall application mapping to AsAP2, for code lengths (a) 4095 and (b) 16129. Update Variable lanes are replicated in the upper left (shaded blue), Compact Set lanes are replicated in the upper right (shaded green), address generators (shaded orange) and data routing cores (shaded cyan) are clustered around the memory modules on the bottom.

2.3.3 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 Figure 2.5. 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 this is not done here, the repacking must be done in a more expensive fashion at the memory interface core.

2.3.4 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 Figure 2.6. 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.

2.4 Results

The computation blocks described in Section 2.3 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 Figure 2.7, 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 Figure 2.8, where cores are grouped into categories. The Correct Variable and Variable Memory Control cores show the largest 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

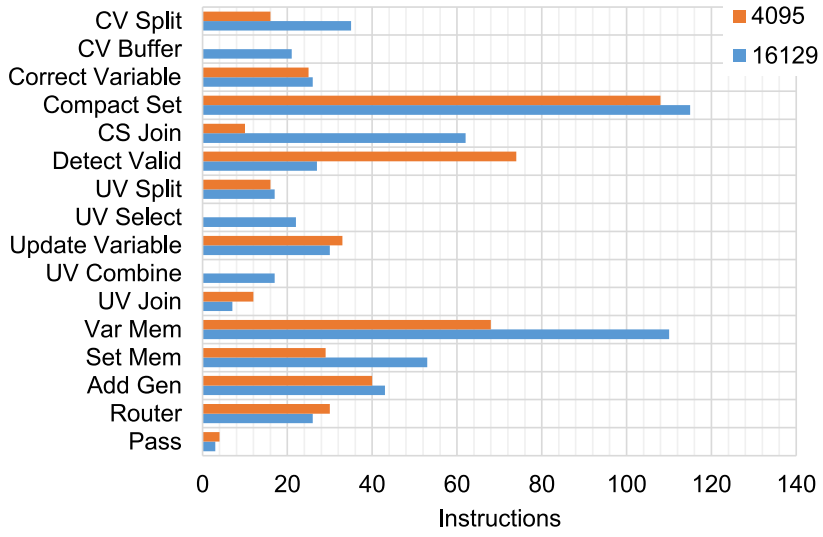


Figure 2.8: Instructions used, given as the highest for any core within a category, for code lengths 4095 and 16129. Cores are limited to 128 instructions.

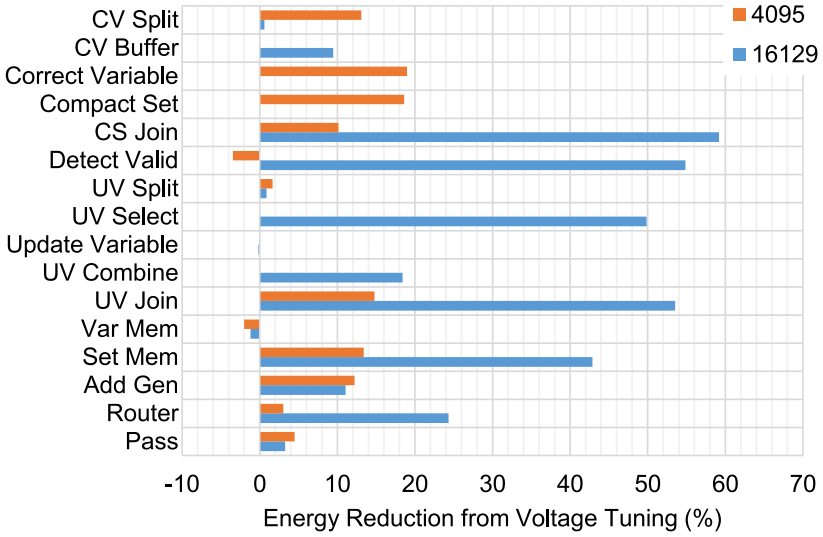


Figure 2.9: Percentage by which energy usage is reduced by optimization, given as the average for cores within a category, for code lengths 4095 and 16129. Reductions are achieved through optimization of the frequency and voltage rail selection for each individual core. Some cores increase in energy due to induced stall cycles from mismatched frequency ratios with neighbors.

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. Figure 2.9 shows the average energy reduction for cores in each category. In some situations, energy usage may increase if a low frequency core transmits to a higher frequency core, due to added empty cycles in the receiving core when reading data bursts. Overall, optimization reduces energy usage by 9.3% and 12.1% with a reduction in throughput of 0.56% and 0.44% for code lengths 4095 and 16129 respectively.

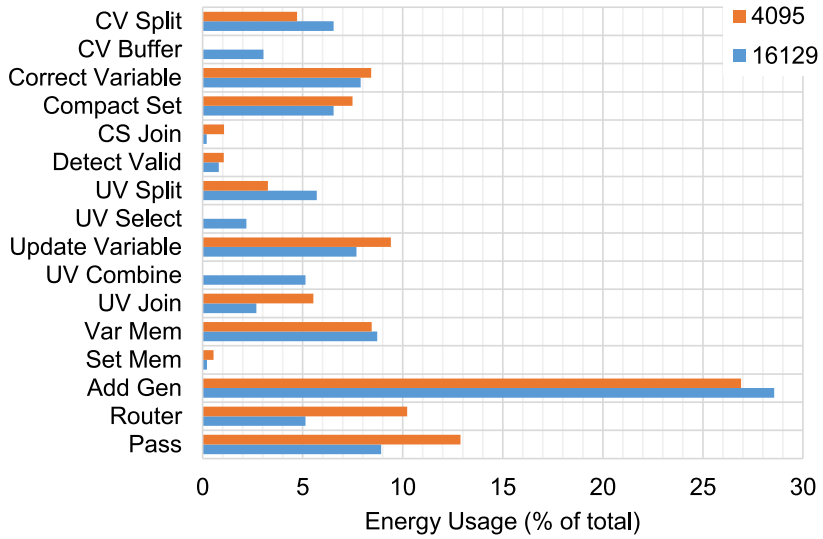


Figure 2.10: Percent of the total application energy used by cores in each category, for code lengths 4095 and 16129.

Figure 2.10 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 2.3. First is a multi-threaded decoder written in C++ using the algorithm described in this work, running on an Intel i7-3770k processor. Second is a GPU decoder presented by Li et al. [26] running on an Nvidia GTX 580. Designs are scaled to 22 nm using the scaling equations presented by Stillmaker et al. [27], 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

Table 2.3: 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 22 nm 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 [26]	2304	40	710.0	1.365	5.82
GTX 580 [26]	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

metrics of interest are throughput per area and bits decoded per unit of energy. When scaled to the same technology, the AsAP2 platform offers throughput per area up to 2.15x higher than the GPU and 75x higher than the CPU, with energy efficiency up to 2.14x higher than the GPU and 77x higher than the CPU.

Chapter 3

From AsAP2 to KiloCore

Many of the architectural features of KiloCore originated in simulation and analysis of AsAP2 and its applications. During work on these applications, a number of insights were achieved into how the AsAP2 architecture could be significantly improved upon. While KiloCore is a freshly written design, reusing none of the AsAP2 implementation code, it does inherit many of the architectural characteristics. The following sections detail many of the major differences between the architectures.

3.1 Applications Explored

A custom conversion tool was used to translate several AsAP2 assembly applications into a format suitable for high speed simulation. This simulator is described later in section 6.1. These applications are described in section 1.2.2, and include the 802.11 baseband receiver, two variations of the AES engine, two variations of Snakesort, and two variations of H.264 video encoding.

Supplementing the above are several fresh applications. The 4095-bit LDPC decoder is described in chapter 2. The High Speed Uplink Packet Access (HSUPA) protocol for 3G mobile telecommunication networks is implemented using 8 processors and one memory. A software floating point suite includes addition, subtraction, multiplication, and division, each occupying one processor.

3.2 Unsigned arithmetic

In AsAP2, all arithmetic operations were assumed to be signed (two's complement) to reduce processor area, with corresponding sign extension in the hardware. In practice, this proved to be

significantly detrimental as many important applications rely on unsigned operations. Three general types of scenarios arose: 1) comparison of unsigned values experienced corruption of the carry bit which prevents direct comparison; 2) adding multi-word signed or unsigned values also experienced carry bit corruption between words; and 3) multiplication of unsigned values, or of multi-word signed values, was heavily corrupted by sign assumptions.

In each of these cases, significant cycle counts were added to obtain the correct results, eg. converting 16-bit values into a longer series of 15-bit values and manually handling carry-out. In addition to cycle and energy overhead, the extra instructions placed substantial pressure on a processor core's limited instruction memory, to the point where other computation kernel optimizations would be omitted due to lack of memory space. The new architecture adds unsigned add, subtract, and multiply-accumulate operations to address these problems.

One of the primary benefits of unsigned support is the improved handling of multi-word, greater-than-16-bit values, such as 20-byte record keys for Sorting applications and 24-bit mantissas in software single-precision floating point algorithms. This also benefits branches which check the ALU carry flag, such as "branch if A is greater than B" where A or B are unsigned values using a full 16-bit range. This compare-and-branch case requires two instructions with unsigned support versus 9 instructions previously. The speedup of several basic operations on an otherwise unmodified AsAP2 are given in Figure 3.1, where a factor of 1 represents no gain, 2 represents computation time cut in half, and so on. The greatest benefit is in 16-bit unsigned multiplies, which require 15 cycles on unmodified AsAP2, and only one cycle with this change.

Signed multi-word multiplies continue to show slowdowns with simple unsigned support due to the half-signed nature of several partial products. That is, for the signed operation C=A*B, when computing the partial product of the high word of A and a lower word of B, a signed value must be multiplied with an unsigned value. Consideration was given to half-sign multiply instructions for KiloCore, but they were omitted due to the number of opcode permutations required. The metrics in Figure 3.1 handle signed multi-word multiplies by operating on the absolute values and separately computing the sign. These signed multi-word multiplies have not been encountered in the target applications.

The reduction in instruction count for each of these computations is approximately inversely proportional to the speedup, but not strictly so due to pipeline hazards inducing no-ops. Figure 3.2

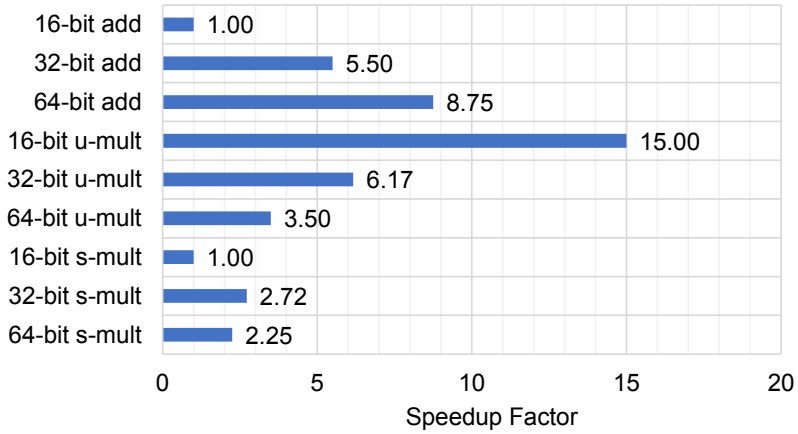


Figure 3.1: Benefit of adding unsigned instructions to AsAP2 for various integer operations of varying sizes. A speedup factor of 1 signifies equal performance. Add applies to signed or unsigned addition or subtraction. Multiplies differ between unsigned (u-mult) and signed (s-mult).

shows the amount of a processor core's 128-word instruction memory that is used by these computations, in the original AsAP2 and with the addition of unsigned instructions.

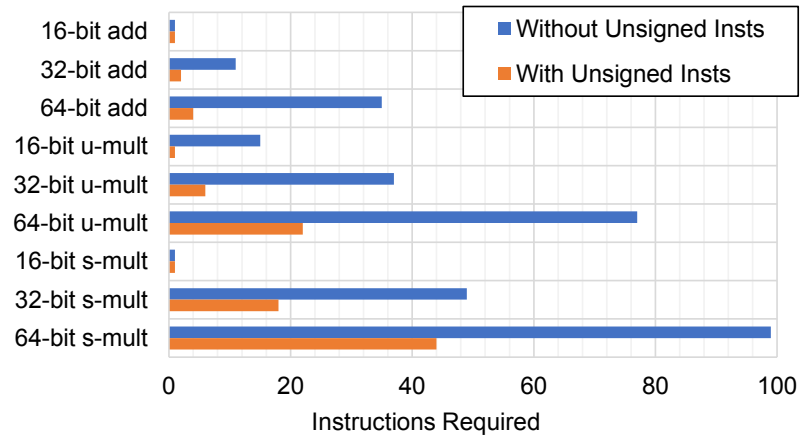


Figure 3.2: Instructions required for various integer operations on AsAP2, with and without the addition of unsigned instructions.

The ALU flags "overflow", "carry", and "negative" naturally support multi-word operations without modification, since they are set based on the final, high word of a computation result. The "zero" flag is inaccurate, as it only indicates the high word being a zero value. Consideration was given to a special "sticky zero" flag that would update across multiple result words, but it was omitted due to lack of common use, awkwardness to implement in the instruction set, and due to extending the ALU critical path.

3.3 Carry-Shift

Added in KiloCore are the SHLC, SHRC, and SRAC instructions. These are shifts that support the current ALU carry bit being appended to the LSB or MSB position (as appropriate) of the pre-shifted value prior to the shift operation. This offers efficient support for multi-word shifts of one bit position, such as occur in CRC calculations, bitwise division and square root algorithms, and mantissa normalization for software floating point algorithms. To support this, the standard shift instructions have also been modified to set the carry bit appropriately. Although shifts of only one bit may at first sound somewhat limiting, they are extremely useful and require a very small amount of additional hardware.

The new instructions added are:

- SHLC : Shift left with carry in
- SHRC : Shift right with carry in
- SRAC : Shift right arithmetic with carry in

Figure 3.3 shows the performance gain in software floating point operations. The longest path measurement is for the worst case cycle count for a single operation, one where the combination of inputs require the longest rounding and re-normalization effort. Overall, the average operation is sped up by 3%, and the longest path is reduced by 6%.

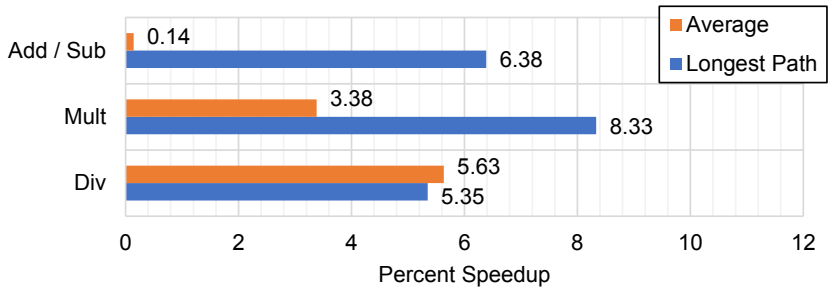


Figure 3.3: Benefit of shift-carry instructions for software single-precision floating point Add/Subtract, Multiply, and Division, when added to an otherwise unmodified AsAP2. The average speedup is across random inputs, whereas the longest path speedup is for input combinations that require the greatest rounding and re-normalization effort.

3.4 Branch unit

In AsAP2, the branch evaluation logic has limited hardware and relies on flags set by the ALU on a previous cycle to implement common branch types. For instance, "if A>B" translates into

two instructions: a subtraction followed by a branch on the carry-out flag. In AsAP2, this meant a branch evaluated in stage 2 depended on an ALU flag setting instruction in stage 4 to have completed, requiring a two cycle delay between these instructions. In actual applications, these delay slots were typically filled with empty cycles due to limited opportunities to schedule useful instructions in that window.

All AsAP2 branches are effectively predicted not-taken, where the immediate instruction after the branch would be fetched automatically but then discarded on a taken branch, a loss of one cycle and related energy.

To address these weaknesses, KiloCore branching differs from AsAP2 as follows:

- The primary branch evaluation logic is placed in pipeline stage 4 instead of stage 2.
- Preliminary branch prediction is included in the program control unit, and is evaluated the same cycle as instruction read with the help of a static prediction bit added to branch instructions.
- Network input FIFOs reads are determined in stage 4 instead of stage 2, and the KiloCore FIFO design supports zero-latency reads (as opposed to two-cycle latency in AsAP2).
- Address generators in stage 2 store prior state for two cycles, to support rollback on misprediction.

By placing the branch logic alongside the ALU, the typical pre-branch no-ops of AsAP2 are eliminated. However, this change results in a greater branch misprediction penalty, which increases from 1 to 3 cycles. The cycle penalty can be largely mitigated by improved branch prediction to limit the frequency of misprediction events. The KiloCore approach allows branch instructions to be statically set as predict-taken or predict-not-taken based on the specific branch opcode used. This requires negligible hardware overhead, especially when compared to dynamic branch prediction units in coarser-grained processors.

AsAP2 applications were found to have highly predictable branches. Figure 3.4 shows the increase of correct branch prediction in several AsAP2 applications after profiling was applied to tune each branch instruction. The lowest gains were found in LDPC, which makes heavy use of conditional execution instead of branching in its computation kernels. Overall, the average prediction accuracy was increased from 27% to 96% with this modification. The net speedup for these branch changes, including the elimination of pre-branch empty cycles and the reduction in

misprediction penalties, is shown in Figure 3.5. Here, the AsAP2 model is modified to match the branch logic of KiloCore, and applications updated accordingly. These measurements were made for each individual core within an application for the speedup of its local program when the core is not stalled on network reads or writes. On average, programs see a 24.7% speedup.

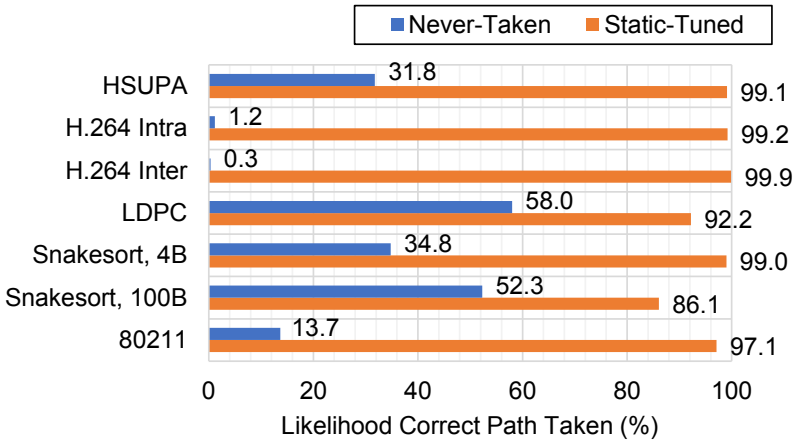


Figure 3.4: Branch prediction success rate for AsAP2 applications, either following the untaken path (original AsAP2) or modified to predict using a static flag per branch instruction selected based on offline application profiling.

Consideration was given to combined ALU+branch operations, enabled by the branch unit and ALU being located in the same pipe stage, to achieve single cycle branching without significant additional hardware. However, even the simple "branch if negative" and "branch if equal" instructions were found to substantially increase the critical path of the processor, so they were removed from the final design of KiloCore.

3.5 Repeat Loop Modification

The repeat instruction allows for low overhead code loops through use of automatic loop hardware. In AsAP2 this was implemented using two counters: one indicating the number of instructions inside the loop, and a second for the number of loops to perform. This imposed a natural limitations that the loop needed to always contain a fixed number of instructions, and that the loop could not be terminated early (eg. "break" or "continue" constructs).

KiloCore's repeat instruction forgoes instruction counting in favor of forming a soft code link between the last instruction and first instruction of a loop, automatically detecting and overriding the post-loop instruction address with the start of loop address. Conceptually, the hardware adds an implicit, 0-cycle decrement-and-branch operation after the last instruction

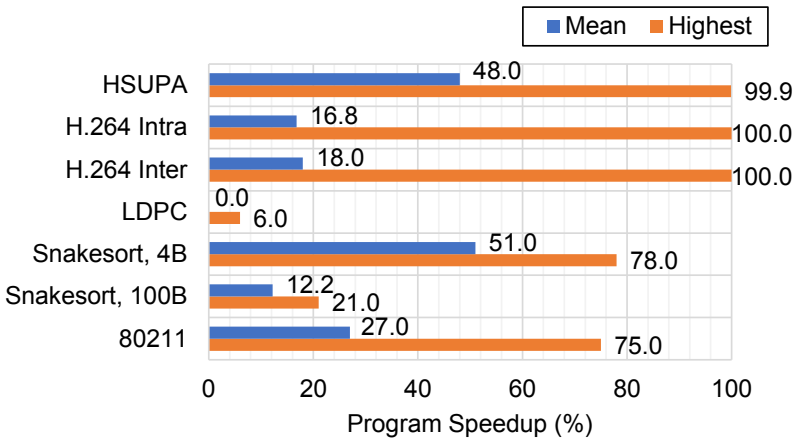


Figure 3.5: Program speedup of AsAP2 applications when branching is modified to behave as in KiloCore, given as the average or max across processors in the application. A speedup of 100% implies an execution time reduced to half. By coincidence, three applications contain processors with speedups of very close to 100% (but not exactly).

of the repeated block. This allows for dynamic code inside of loops without the loop hardware adding complexity to the program flow, and also allows early loop termination by branching out of the loop block without returning.

A secondary benefit of this change is reduced overhead for starting repeat loops. In both the old and new implementation, there is a 3 cycle delay between launching a repeat instruction and the Program Counter unit being configured for repetitions. In AsAP2, since the repeat loop needed to count instructions inside the loop, this 3 cycle setup delay needed to be filled with 3 no-ops so that the PC unit was configured before the first loop instruction was launched. In KiloCore, this 3 cycle hazard is between the repeat instruction and the last loop instruction, allowing the hazard to be filled by earlier loop instructions.

3.6 Network IO

3.6.1 Stall on Multiple-I/O Simultaneously

In AsAP2, if a core needed to handle dynamic data flows where the ordering of data was not well defined, it needed to enter a polling loop to continually check the appropriate FIFO status signals. For example, a core might loop between checking input FIFO 0 and input FIFO 1 in order to respond to the first value to arrive on either input. These loops required high energy consumption when no practical work was being performed. Such loops occur in the H.264 application.

The addition of new STALLIN and STALLOUT instructions allow for stalling a processor core until one of a selected set of inputs or outputs becomes not-empty or not-full respectively. A stalled core will turn off its local oscillator to reduce active power to zero while waiting. The new stall instructions take in, as a source argument, a bitmask of input or output directions to wait on. After a program has moved past a stall instruction, a natural next step would be for the core to perform one or more conditional branches to determine which input or output direction is then ready for use.

3.6.2 Explicit output buffer destinations

In AsAP2, a processor would select a set of desired output directions by modifying a special bit mask register, where output writes are sent to all selected directions. Stall logic would check for full output FIFOs during stage 2, while the direction register would be written during stage 6. As a result, any output writing instructions needed to be launched 5 or more cycles after an output direction change for safe operation. This imposed a large penalty on applications that required frequent changes in output direction.

In KiloCore, each individual output direction is available as an instruction destination, in addition to a broadcast register that behaves like AsAP2. This allows programs to cleanly transition between single-direction writes, which make up the large majority of writes to be performed. When a multiple-direction write is needed, changes to the broadcast register have been streamlined: the stall logic module has been moved to stage 4, and the output direction mask has been assigned a dedicated register written during stage 4. This allows a direction mask changing instruction to be immediately followed by an output write, negating the 5 cycle hazard from AsAP2.

3.6.3 Quicker processor clock halting

When an instruction stalls a processor, such as due to reading an empty input buffer, there is a danger of pending output writes from recent instructions still being in transit between the processor and target FIFO, that will become stuck if the processor halts its clock immediately. Such a situation could lead to whole-application lockup if the input the processor is waiting for depends on the stuck output's data, due to some dependency loop in the application. A simple example of this would be a synchronization handshake between neighboring processors. Any data flow that is not purely linear is potentially in danger.

In AsAP2, a processor would wait for a fixed 8 cycles after a stall occurred before its clock would halt, allowing output writes to exit the processor pipeline and also traverse registers in the network and the target processor's FIFO to complete a write. In most cases, this inserted excess stall cycles, as typically few if any cycles were necessary to complete output writes before halting the clock. Furthermore, this fixed cycle count still had a danger of writes becoming stuck when several network registers were enabled in a long distance circuit network link.

In KiloCore, this situation is addressed by dynamically adjusting the number of inserted stall cycles based on how recently an output write was launched. In hardware, this is achieved using a counter to track the progress of writes. When an output writing instruction passes the stall logic check in stage 4, it sets the countdown to a configurable value (set statically during programming). Each cycle the count decrements towards 0, at which point the processor may halt its oscillator if there are no other stay-awake conditions. If no output writes are pending when the processor reaches a stall, as is commonly the case, it may halt immediately. Each individual processor may be configured for a different number of write cycles based on the number of registers in its output circuit network link, along with the pipeline depth of instructions which write to the network.

In applications where the computational loads between processors that communicate with each other are frequently mismatched, the number of stall events can be very high, contributing significantly to the total application energy. Figure 3.6 shows the reductions in output stall cycles and total energy for several applications when this architectural change is implemented in AsAP2. On average, stall cycles with an active clock are reduced by 33%, and overall application energy usage reduced by 3%. Individual applications see up to a 75% stall cycle reduction or up to a 7% energy usage reduction.

3.6.4 FIFO Depth

In AsAP2, the inter-processor communication FIFOs have a depth of 64 16-bit words, with two such FIFOs per processor. These FIFOs serve two purposes: 1) buffering so that the sending processors can continue its program without waiting on the receiving processor to consume data, and 2) storing write data during the latency cycles between the sender starting a write in its pipeline and the distant receiver FIFO's "full" signal being updated and returned. To explain the latter case: when the sending processor starts a data write which will eventually switch the receiver FIFO from "not-full" to "full", there will be a number of cycles of delay before the write

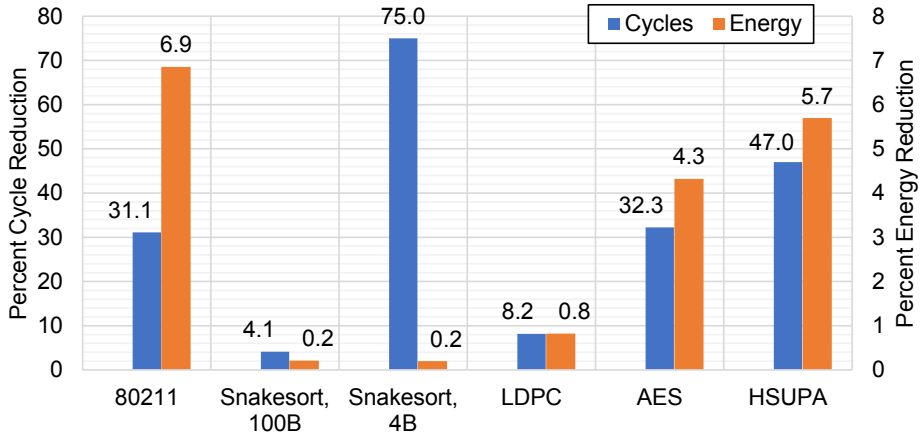


Figure 3.6: Energy reductions and stall cycle reductions for AsAP2 applications when changing the processor stall logic to halt as soon as network writes are flushed from the pipeline. A stall cycle is when a processor's program is paused due to a data dependency, but the processor's oscillator is still cycling the clock.

completes, the FIFO status updates, and this new status is returned to the sender. During this window, the sender is allowed to continue to transmit write data to the FIFO, and the FIFO is configured to signal being "full" prior to actually being full, such that the reserved space can safely capture any pending writes before the sender pauses transmission.

Several AsAP2 applications were simulated with varying FIFO depths and had their throughput measured. In Figure 3.7 FIFOs were modeled with a simplified zero-latency network and no reserved space. This captures the application's baseline benefit from a transmitting processor continuing its program after initiating a network write, without having to wait for the receiving processor to consume the data, and only pausing once the FIFO is full. Metrics are normalized to a 512 FIFO depth. Both versions of AES cease to function below a depth of 8 due to internal inter-processor dependency loops, and 802.11 is similarly nonfunctional below a depth of 32. Snakesort for 100-Byte records sees large penalties for FIFOs below a depth of 64 words (128 Bytes), as a processor can no longer fully store an output record in a FIFO and must pause for the receiver. These behaviors could potentially be adjusted with changes to the application source code.

In Figure 3.8 this same analysis is performed using real network latency and a 10 word reserve space, the default in AsAP2 applications. Most applications were found to have little change in throughput beyond 32 words, with the exceptions of HSUPA and Snakesort of 100-byte records.

For KiloCore, a 32-word FIFO depth is selected. To address the Snakesort penalty, the KiloCore rewrite of the Snakesort kernel splits each record in two halves to be transmitted over two links, sending 25 words (50 bytes) to each FIFO.

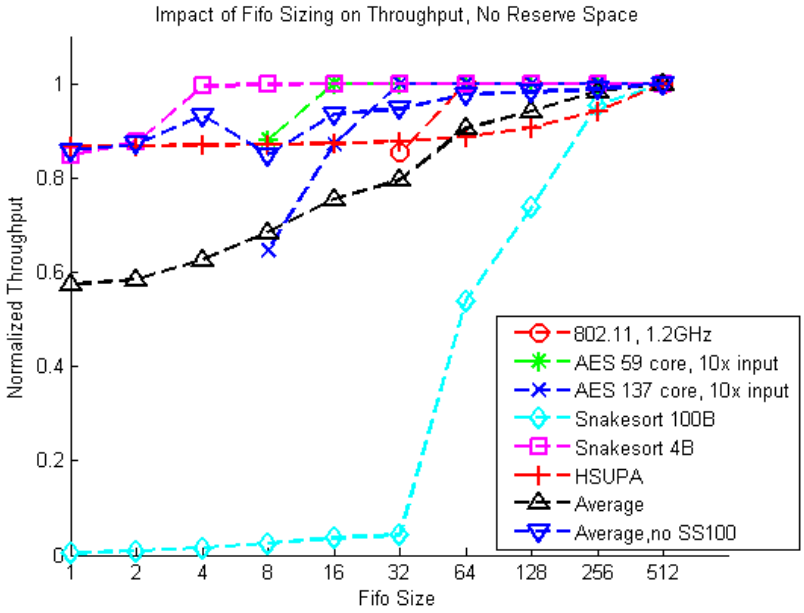


Figure 3.7: Impact of inter-processor communication FIFO depth on AsAP2 application throughput, modeled without write latency or corresponding reserve space. Metrics are normalized to a 512 FIFO depth. This captures the application's baseline benefit from a transmitting processor continuing its program after initiating a network write, without having to wait for the receiving processor to consume the data.

3.7 Address Generator Modification

Each AsAP2 processor contains a set of address generation units which provide automatically incrementing pointers. These are typically used to optimize the traversal of data blocks and occasionally see use in other optimizations.

AsAP2 provides four such address generators, but the typical program used only one if any of these. The highest usage peaked at three, with the fourth address generator providing no benefit to the examined applications. Furthermore, the previous address generators contained AND and OR masks to set or clear their address bits, a reverse stride setting, a bit reversal and automatic right shift option, and a shift mask which caused selected address bits to be shifted left and OR'd with their unshifted neighbors. In the examined applications, none of these special features were used.

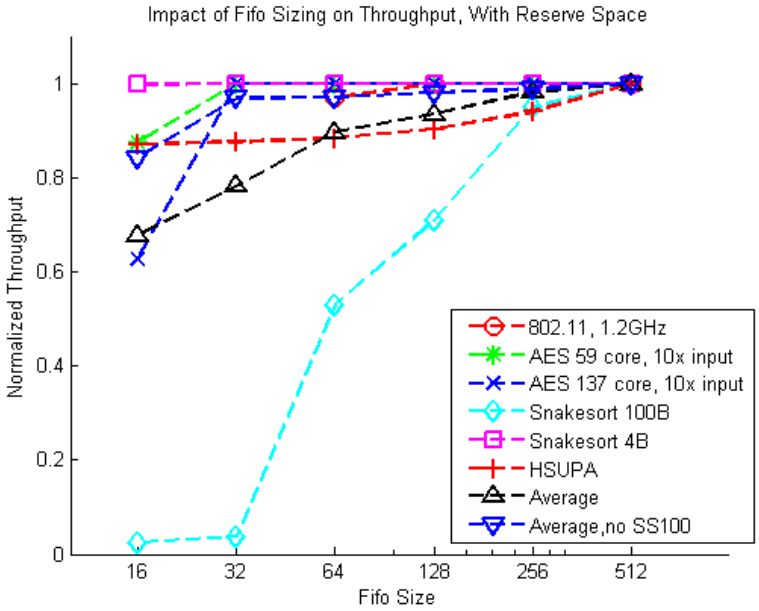


Figure 3.8: Impact of inter-processor communication FIFO depth on AsAP2 application throughput, with realistic write latency and reserved FIFO space to safely prevent overflow. Metrics are normalized to a 512 FIFO depth.

KiloCore implements three address generators, the maximum usable by any given instruction (one dest and two sources), and reduces the configuration options to just the start address, end address, and a signed value for the stride.

3.8 Parallel Data Memories

A challenge experienced during both the AsAP2 and KiloCore tapeouts is the lack of available full speed, three-port (1 write, 2 read) SRAM designs from the foundry for use as data memory. While a double-pumped variation is available, its operational frequency is cut in half, removing it as a practical option in a high performance design. It is possible to settle for a two-port SRAM, but this requires instructions that read two memory locations to be split across two cycles, imposing penalties on performance, energy efficiency, and potentially instruction memory usage.

To determine the importance of instructions that obtain both source operands from data memory (as opposed to network inputs, constants, pipeline bypasses, etc.), the prevalence of double memory reading instructions in several sample applications was examined and is shown in Figure 3.9. Here, the software floating point kernels are collected together under the FP entry. Analysis is performed on a per-processor basis, and determines what percentages of instructions

in that processor perform dual reads. The figure shows the average across all processors, and the processor with the highest percentage. While the average program across applications is made up of 7.8% of such double-read instructions, the peaks can be very high, up to 54% in 802.11.

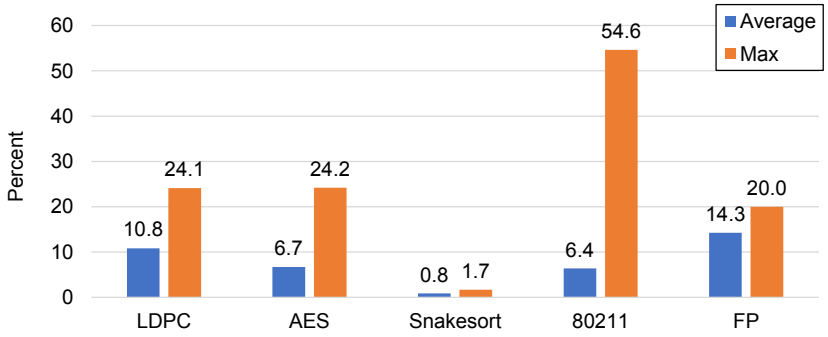


Figure 3.9: Percentage of instructions requiring two data memory reads in AsAP2 applications, given as an average across all processors and as the highest processor.

To work around this challenge, AsAP2 utilized a pair of dual-port SRAMs with write ports tied together, creating an effective triple port memory but at the cost of doubling the area and write-back energy. During the design of KiloCore, the data memory was instead broken into two independent banks, each of which can be instantiated with a single dual-port SRAM with individual write-back. For nearly the same area, this effectively doubles the maximum available data memory, but imposes coding challenges for variable to memory mapping. A method of addressing these challenges is discussed below.

Ultimately, the final KiloCore design opts for flip-flop based memories to achieve higher performance, bypassing the SRAM restriction on three ports. However, this two-port dual-bank approach is still utilized. When comparing synthesis of a single three-port flip-flop memory bank to a pair of two-port memory banks of equal combined capacity, the dual-bank solution occupies 16% less area, reduces active access energy by 46% for single-bank writes, and reduces leakage power by 24%. The hardware logic implementation is discussed further in chapter 4.

3.8.1 Software adjustment

With instructions that can read from two source operands, if both of the sources reference variables assigned to the same memory bank, an error will occur since the bank is only able to source one of the variables through its single read port. There are three straightforward ways to address this situation: 1) spend an extra cycle to pre-read one variable before each instruction

that needs to read a single bank twice; 2) write one or both of the conflicting variables to both banks; or 3) reassign one of the variables to the other memory bank at compile time.

When converting existing applications that were written for a single, dual-read data memory to use two single-read memories, the most straightforward method is to assign all variables to be written to both memories at write-back (called a Dual assignment). In this case, the two single-read memories are functionally equivalent to the previous architecture's dual-read memory. However, this also requires the greatest amount of data memory to support the applications. A basic optimization is to only assign variables as Dual if they appear in an instruction that reads two variables; all other variables appear alone and may be assigned to either memory freely. The Dual and Single variable breakdown for several AsAP2 applications is shown Figure 3.10, using this basic assignment method. On average, 18% of variables are mapped to both banks.

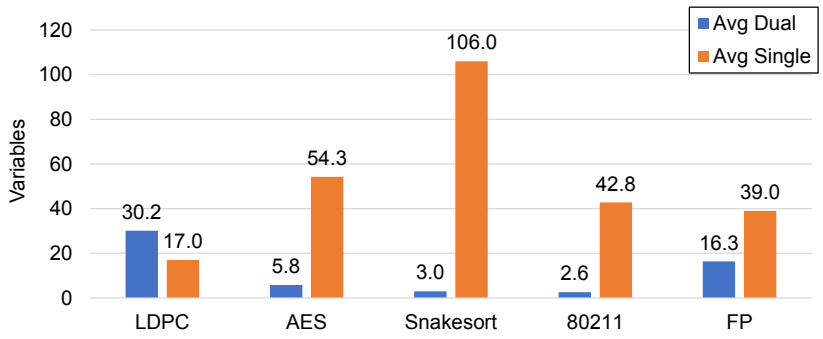


Figure 3.10: Pre-optimization number of Dual and Single write variables, averaged across processors, in AsAP2 applications converted to utilize two single-read-port memory banks. Dual variables must be written to both banks to avoid software slowdown.

Using profile data for each application, variable assignment to low, high, or both memory banks can be optimized. By reducing the number of double writes, more memory becomes available for use and write-back energy costs are reduced. Without performing any changes to the existing code beyond memory location reassignment, Algorithm 1 was applied to the program for each core.

The results of this optimization step are shown in Figure 3.11, where nearly all Dual variables are eliminated. Further optimization is possible with refinement of the algorithm. For three of the five test applications, dual write backs are eliminated entirely. One core in 802.11 requires two Dual variables, floating point addition requires one, and floating point division requires two. The reduction in average and maximum per-core memory usage is shown in Figure 3.12.

Algorithm 1 Basic algorithm for assigning variables to data memory banks.

1. For each variable, record a list of all neighbor variables—those that are read in the same instruction.
2. Order the list of variables by the number of neighbors, descending.
3. For each variable:

3a. If assigned as Dual, skip.

3b. If unassigned: if any of its neighbors is assigned to Low, assign this variable to High. Otherwise, assign this variable to Low.

3c. For each unassigned neighbor, assign it to the opposite data memory (eg. Low if this variable is High).

3d. If any neighbor is already assigned to the same data memory as this variable, a conflict exists.

4. If no conflict was found, optimization is complete. Otherwise:

4a. Out of all non-Dual variables, assign the one with the most neighbors to Dual.

4b. Reset all other non-Dual variables to unassigned.

4c. Repeat step 3.

LDPC in particular benefits from its peak usage falling below 128 words, allowing it to run at a smaller memory stepping. All applications would operate with two 64-word memory banks, though KiloCore opts for the larger 128-word banks for future application development.

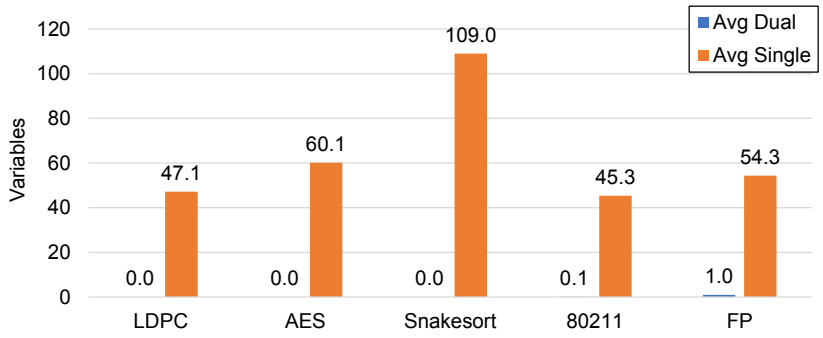


Figure 3.11: Post-optimization number of Dual and Single write variables, averaged across processors, in AsAP2 applications converted to utilize two single-read-port memory banks.

3.9 Voltage Tuning

Many-core applications typically exhibit significantly varying workloads across cores, and substantially benefit from reducing per-core voltages when it will not impact application throughput. The following analysis focuses on static frequency and voltage assignments for cores based on application profiling. This approach offers substantial energy savings with little hardware

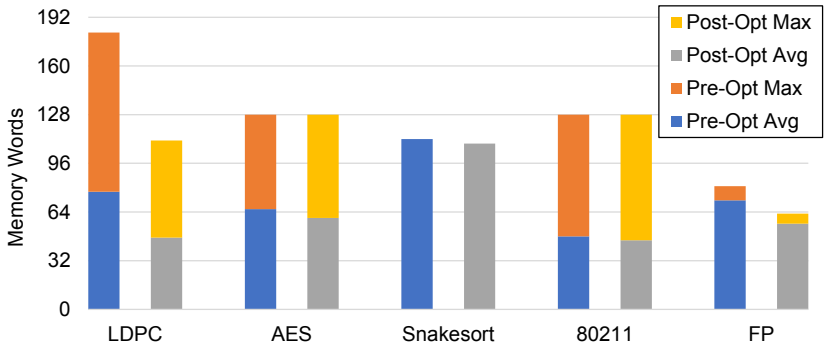


Figure 3.12: Total data memory usage of a processor, given as peak and average across processors in AsAP2 applications converted to utilize two single-read-port memory banks, both pre- and post-optimization.

overhead per-core. These static assignments can be determined for a variety of full-application performance points and used in an array-wide Dynamic Frequency and Voltage Scaling (DVFS) scheme, but this analysis only focuses on optimizing individual performance points.

3.9.1 Application Profiling

Central to optimizing voltage is knowledge of the required operating frequencies of each core in an application, as the primary constraint on voltages is that they are high enough to support these frequencies. However, determining these frequencies is not trivial. In many-core applications, workloads of the cores do not correlate to the necessary operating frequencies. Inter-core dependencies create complex workload patterns, such that even very low workload cores may be required to operate at high speeds if they contribute to an overall application critical path.

A reliable way to determine these optimal frequencies is through iterative profiling. In this, an application running a representative workload is iteratively simulated under varying per-core frequency conditions, starting with all cores set to a frequency that achieves the application performance target. For each core in the application, its individual core frequency is gradually reduced until a drop in overall application throughput is observed, and the pre-drop frequency estimated as the core's ideal frequency.

In practice, even if the ideal frequencies do not impact application throughput, they will increase latency, and short simulations cannot easily separate latency from throughput. As such, some allowance is made for the simulated throughput estimate being reduced by frequency tuning. In the rest of this analysis, the final throughput reduction has been constrained to less than

3%, where each individual core tuning may reduce throughput by a fraction of that percentage depending on the number of cores in the application.

When tuning is complete, the application profile will contain the minimum required frequency for each core in the application. Arbitrarily complex core interactions are accounted for in these frequencies, contingent on the interaction being present in the representative workload.

3.9.2 Voltage Model

This analysis will make use of a voltage model constructed using measured data from AsAP2, which was fabricated in 65 nm CMOS. Due to low core areas and high core counts, providing unique voltage domains to each core is impractical: an on-chip voltage regulator is much larger than a given core, while off-chip delivery will not scale for potentially 1000+ cores requiring unique voltages. The chosen approach is to use a limited set of shared voltage domains, with a given core connecting to a single domain at any given time. AsAP2 made use of two such voltage domains; this analysis is aimed at finding the ideal domain count for real applications. Where appropriate, the voltage model includes energy spent or recovered when switching a core between voltage domains.

3.9.3 Voltage Selection

The voltage for a given core may be reduced to the minimum that will support that core's profiled frequency requirement. Two methods are explored for assigning voltages: 1) static assignment, where each core connects to a single voltage rail; and 2) dithered assignment, where a core will alternate between two voltage rails on a fixed schedule.

Dithered assignments allow a core to alternate between a low and high frequency such that it achieves the required effective frequency, but saves energy for those cycles spent on the lower voltage rail. In hardware, this requires setting the number of active cycles to spend on each rail, with a counter triggering an automated rail switch when the desired cycles are reached.

Voltage selection consists of two parts: picking a voltage for each available rail, and assigning individual cores to these rails according to their required effective frequencies. The target optimization goal is to find the rail voltages which provide the greatest application energy savings.

Energy and voltage have a non-smooth relationship, where incrementally increasing the voltage of a rail will increase energy use for all cores attached to this rail, but may allow cores that were previously operating at a higher voltage to decrease energy by dropping to this rail.

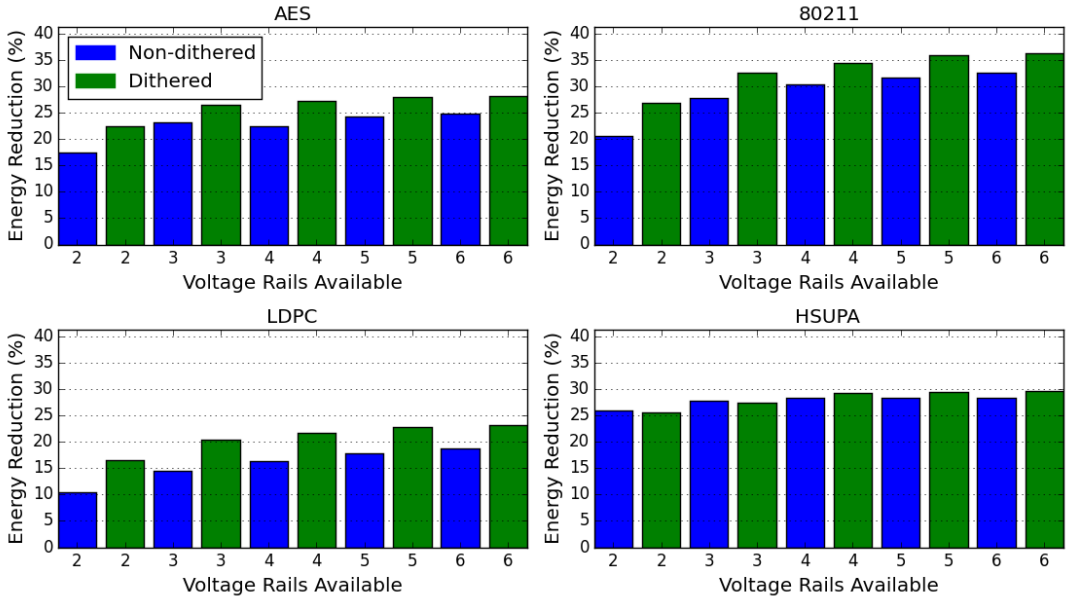


Figure 3.13: Reduction in energy consumption due to voltage optimization in several AsAP2 applications, across varying numbers of available voltage rails, for dithered (alternating) and non-dithered (static) per-core rail assignments.

The chosen optimization algorithm consists of checking every possible voltage combination with a coarse granularity, then reiterating at increased granularities but only checking within a narrow window of a rail's previous voltage. The final granularity is 1% of the maximum operating voltage.

3.9.4 Analysis

Profiling and optimization was performed on several AsAP2 applications, aiming to maintain their performance at the highest operating point. Figure 3.13 shows the expected reduction in energy usage (per unit of workload) for each of the profiled applications. Reductions are in comparison to having a single voltage domain available for all cores. Dithered assignments assume voltage switching happens while a core is active (the core need not add idle cycles during a switch). Figure 3.14 shows the average of the four applications. Energy savings begin with an 18% reduction when allowing two voltage rails without dithering, increasing up to 36% with six rails and dithering support. Dithering gives anywhere from 12% to 23% improved energy savings over the non-dithering cases, averaging a 16% benefit across rail counts.

Figure 3.15 shows the spread of voltages assigned to each rail for each application. Voltages are normalized to the maximum operating voltage. In all cases, at least one rail is assigned to

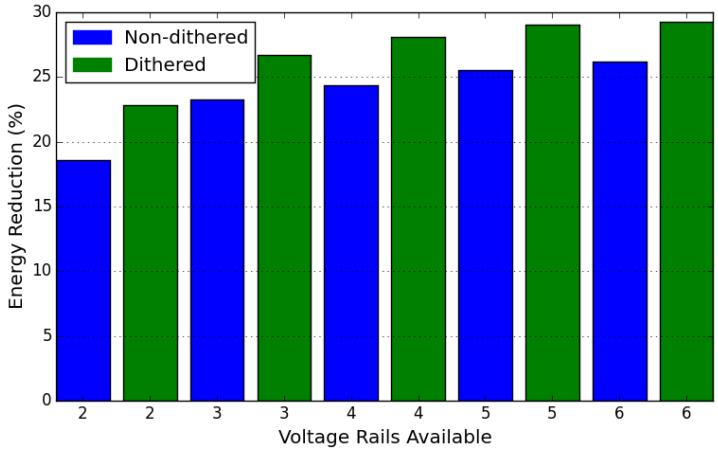


Figure 3.14: Reduction in energy consumption due to voltage optimization averaged over AsAP2 applications, across varying numbers of available voltage rails, for dithered (alternating) and non-dithered (static) per-core rail assignments.

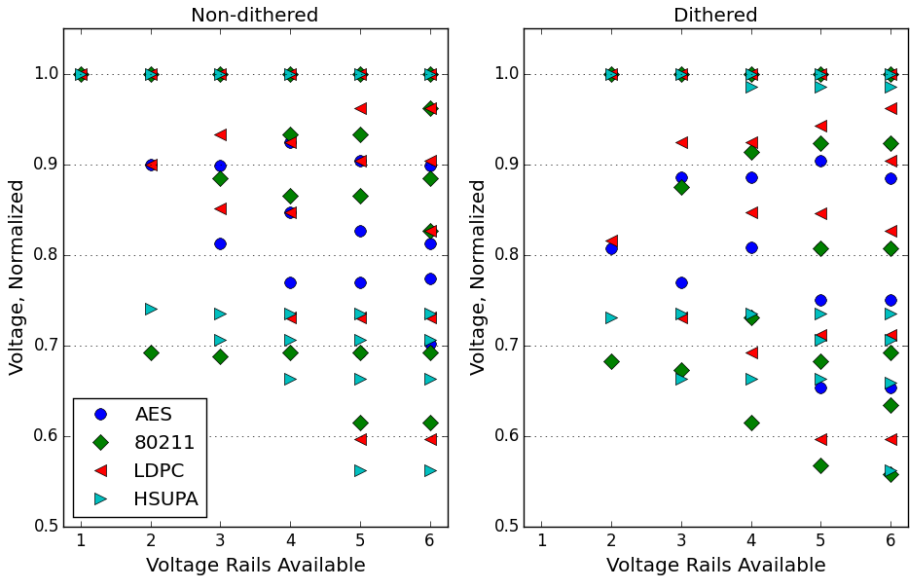


Figure 3.15: Voltages selected for each rail, across AsAP2 applications, normalized to the maximum operating voltage. Voltages are selected to minimize application energy usage.

the maximum voltage in order to support the most critical cores in the application. The spread of voltages increases as more rails are added, with the lowest rail operating down to 55% of the maximum voltage. This spread needs to be accounted for during physical design, to avoid excessive reverse currents through the power gates connecting to each rail.

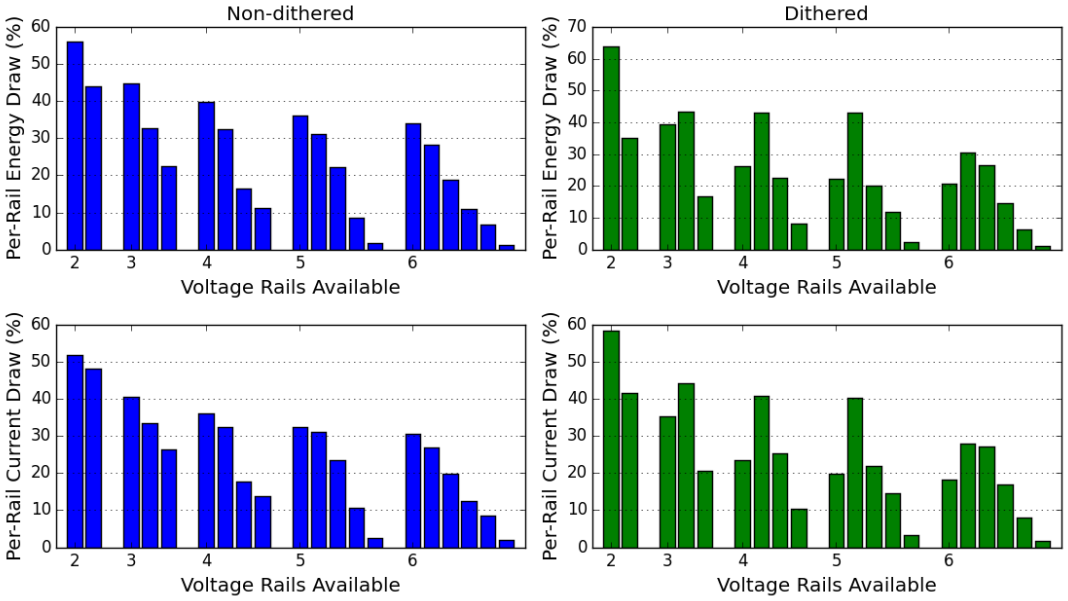


Figure 3.16: Voltage rail load distribution, given as energy draw or as current draw summed across cores attached to the rail, averaged across AsAP2 applications.

Figure 3.16 shows the per-rail load distribution, giving both the energy drawn from the rail as well as the average current. Without dithering, lower voltage rails are always loaded less than higher voltage rails. However, with dithering for three or more voltage rails, the maximum load is experienced by the second highest voltage rail. In this case, cores that would normally be connected to the highest rail constantly are instead spending a portion of their time on the second rail, increasing its load. Load distribution may be considered during physical design when selecting the metal and power gate distribution across the rails.

Figure 3.17 calculates the energy * area product for each rail count; lower is better. Area consists of the additional power gates (1.4% of core area) to connect to a voltage rail with less than 4% peak voltage drop at maximum operating voltage, matching the AsAP2 physical design. Areas are normalized to the single rail case, which is assumed to contain a single set of power gates along with control logic for depowering a core.

The benefits of adding additional rails are found to largely level off after the third rail. Optimums are found at four rails without dithering or five rails with dithering, providing a 0.3% or 0.5% improvement over three rails respectively. KiloCore2 opts for three rails, as discussed further in chapter 5.

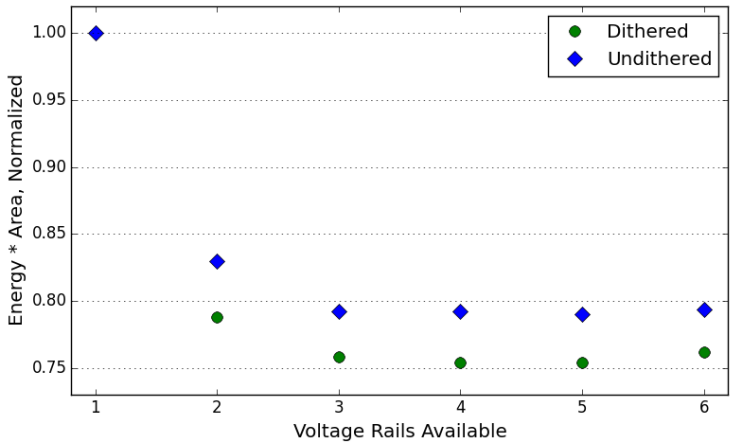


Figure 3.17: Energy * Area products based on the number of available voltage rails, where energy is the amount consumed by an AsAP2 application. Application energies are individually normalized to the 1 rail case, then averaged together. Area represents that required for additional power gates to support connecting to a rail.

3.10 Large Program Support

When an application would benefit from a large program kernel, such as to handle control logic, AsAP2 offers the programmer no convenient way to implement this if it goes beyond the 128 instruction limit of a processor. A potential solution would be to add one or more larger processors to the array, but each of these would occupy the area of many of the standard processors, and would often go underutilized by many applications. KiloCore opts for an alternate solution: using the on-chip shared memories (external to the processors) to supply instructions and program control for these larger kernels.

Since only one version of processor layout is used for all processors in the array, any hardware added for communicating with a memory module will also be added to processors that do not neighbor such a memory module. Therefore, the hardware added to each processor is minimized or shared to reduce this penalty.

Incoming instructions are streamed to one of the 16-bit data FIFOs of the processor, as would a normal data stream. An internal module concatenates these 16-bit words into full instruction words for launching into the pipeline. The pipeline freezes when waiting for an instruction, maintaining correct instruction order and allowing streamed code to be scheduled in the same manner as standard code.

Branchless code may be streamed from 1) other cores, 2) an on-chip memory, or 3) from an external source directly, but branching code is supported only when the core is attached to an on-chip memory.

Program control is handled inside a memory tile, where the program address width is expanded to 16-bits. Control cannot be managed internally in a processor since it only supports 7-bit instruction addresses, corresponding to a local 128 word instruction memory. The processor and memory tiles operate on separate clocks, as part of a GALS architecture, introducing significant latency. The round trip delay for sending an instruction and receiving a branch result is approximately 31 cycles when configured for strong metastability protection (14.5 processor cycles and 12.5 memory cycles, slightly varying based on clock alignments).

The memory instruction stream is provided alongside the standard memory data ports, allowing both instructions and data to be provided by a single memory. To support branches and general program flow, an additional port is added between a processor and the memory for carrying 2-bit response signals. Response signals are sent to indicate a correctly predicted branch, a mispredicted branch, or a miscellaneous executed instruction.

The memory controller will queue a number of instructions to the processor before stopping to wait for responses. The benefits of queuing instructions are shown in Figure 3.18, starting from a minimum of 5 in order to support the sample computation kernels without code modification. The kernels tested are from the Snakesort, software floating point addition/subtraction, and software floating point multiplication. Performance is measured relative to a kernel running out of a processor's local memory, with the peak theoretical performance for streamed instructions being 0.33, or one instruction every 3 cycles when transferred over the 16-bit data bus. Performance is found to peak at 11 instructions queued for this architecture. Since the processor needs to flush excess instructions after a branch misprediction, queuing additional instructions beyond 11 imposes additional energy penalties, and may impose a performance impact if the processor clock speed is much slower than the memory clock speed due to the overhead of emptying the instruction FIFO. This 11-instruction limit may be implemented in the memory-side program controller using a small counter of instructions sent to the processor that haven't yet been acknowledged as executed.

When a branch misprediction occurs, the processor will signal the memory of this event and then begin flushing its instruction FIFO, which is filled with mispredicted-path instructions. The

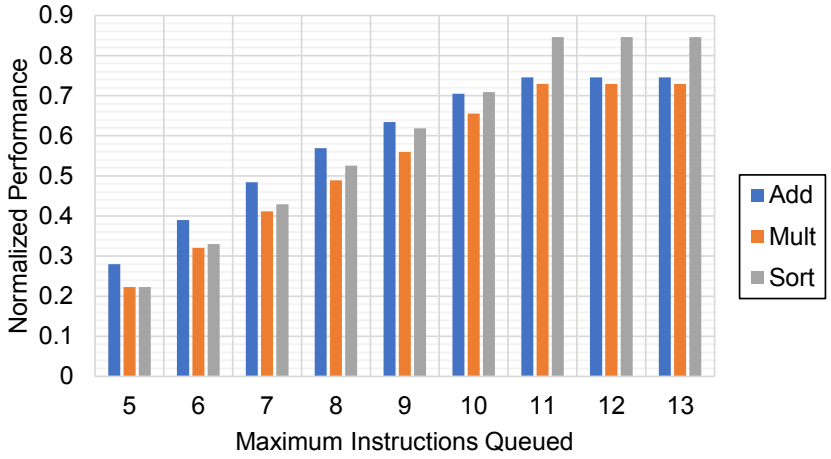


Figure 3.18: Performance benefit of increasing queued instructions when running large programs out of a shared memory module, for the three sampled kernals, normalized to ideal branchless code performance.

memory, upon receiving the misprediction signal, will reset itself to the correct path. A special code word is sent from the memory to the processor to indicate the end of a mispredicted stream and start of a corrected stream. A branch correction queue in the memory holds the alternate paths of recently predicted branches, for supplying the correction addresses to the program controller. Return addresses are also queued for correcting mispredicted Jump style branches.

Figure 3.19 shows how the depth of the branch prediction buffer impacts performance. This buffer depth limits the number of branch paths which will be followed before the evaluation of the first predicted path is returned by the processor. The tested kernels were found to reach a peak performance at 4 layers for Sorting, 5 layers for FP Multiply, and 6 layers for FP Addition. Addition contains the most stressful branching code, up to 8 branches within an 11 instruction window. KiloCore implements a queue depth of 8.

For bootstrapping purposes and to accelerate portions of a kernel, program control may be dynamically transferred between the processor (operating out of its local instruction memory at full speed) and the external memory program at runtime. Special instructions are provided for triggering control handoff in either direction. When in control, both the memory and processor programs are capable of settings each other's instruction address before a control switch, allowing for a variety of program layout optimizations. Conceptually, the program can be primarily housed in the processor and use the external memory for long function calls, or the program can be housed in the external memory and use the processor for fast function calls.

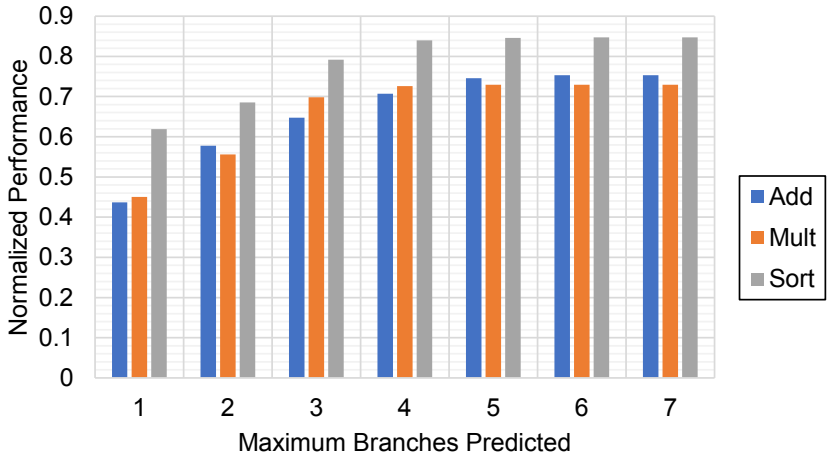


Figure 3.19: Performance benefit of increasing layers of branch prediction when running large programs out of a shared memory module, for the three sampled kernels, normalized to ideal branchless code performance.

Chapter 4

KiloCore

4.1 High-Level Architecture

The KiloCore chip includes 1000 independent, uniform, programmable, RISC-type, in-order, single-issue processors; and 12 independent memory modules [12]. Processors are arrayed in 32 columns and 31 rows with 8 processors and 12 independent memories in a 32nd row as shown in Figure 4.1. Processors and independent memory modules with no work to do dissipate exactly zero active power (leakage only)—this is an important capability in the 1000-processor-chip era due to the difficulty in implementing complex software workloads that spread evenly over thousands of processors which leads to the increasing prevalence of processors with widely-varying activity levels [13]. Under most conditions, the processor array has a near-optimal proportional scaling of power dissipation over a wide range of activity levels.

4.1.1 Processors

Each processor contains a 128 x 40-bit instruction memory, 512 Bytes of data memory, three programmable data address generators, two 32 x 16-bit input buffers, and a 16-bit fixed-point datapath with a 32-bit multiplier output and a 40-bit accumulator. The 72 instruction types include signed and unsigned operations to enable efficient scaling to 32-bit or larger word widths, with no instructions being algorithm-specific. Processors support predication for any instruction using two conditional execution masks, static branch prediction, and automated hardware looping for accelerating inner loops. Although the natural word width of the datapaths and memories is 16 bits, through software other word widths are easily handled—for example, 32-bit floating point [28] and 10-Byte sorting keys for 100-Byte data records [21].

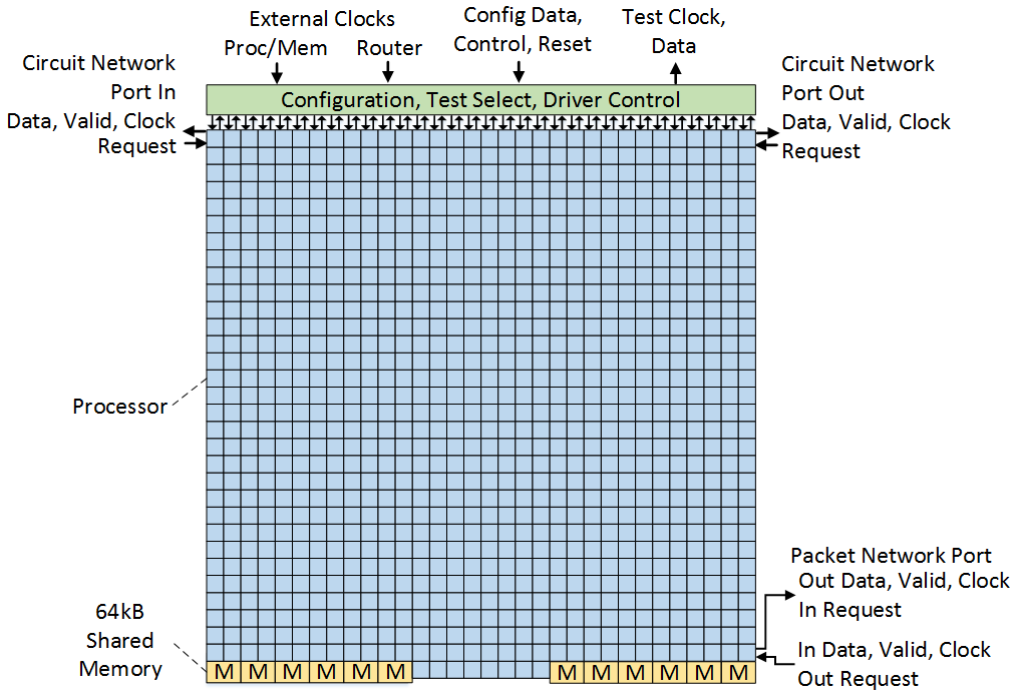


Figure 4.1: KiloCore top-level processor array diagram.

Each processor issues one 40-bit instruction in-order per cycle into its 7-stage pipeline (shown in Figure 4.2) from its local instruction memory and it may also source large programs from an on-die independent memory module. Instruction input operands and output results commonly come from or go to the local data memory, one of several circuit-switched network ports, the packet router port, an attached independent memory, a pipeline forwarding path, or a series of special dedicated-purpose registers that include dereferenceable pointers, address generator configuration, predication flag masks, oscillator frequency selection, and other software-accessible core configuration fields.

4.1.2 On-Die Communication

The processor array connects processors and independent memories via a two-dimensional mesh, a topology which maps well to planar integrated circuits and scales simply as the number of processors per die increases. Communication on-chip is accomplished by two complementary means: a very high-throughput and low-latency circuit-switched network [29] and a very-small-area packet router [30] which is especially well-suited for high fan-in and high fan-out communication; details are provided in Figure 4.3.

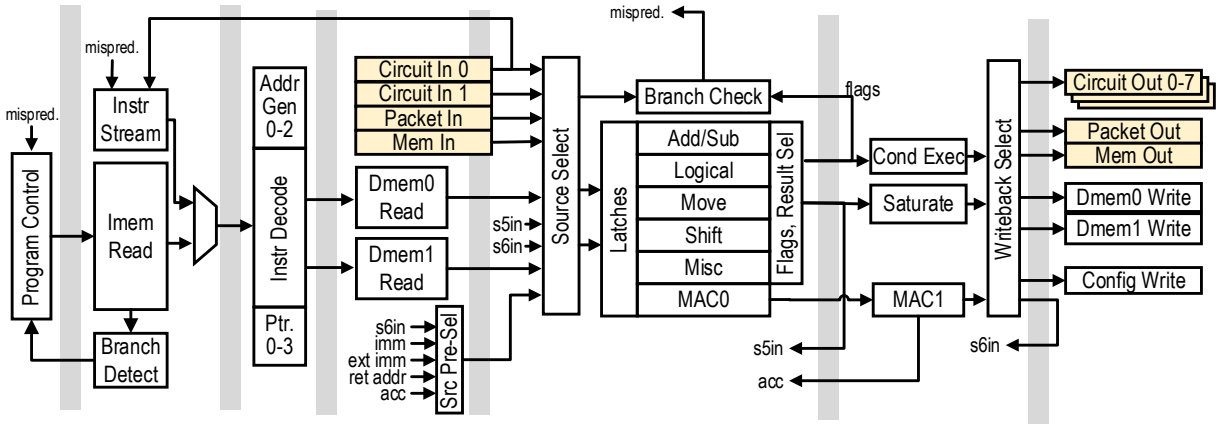


Figure 4.2: Major components and connections of the 7-stage processor pipeline. Several control and configuration signals are omitted for clarity.

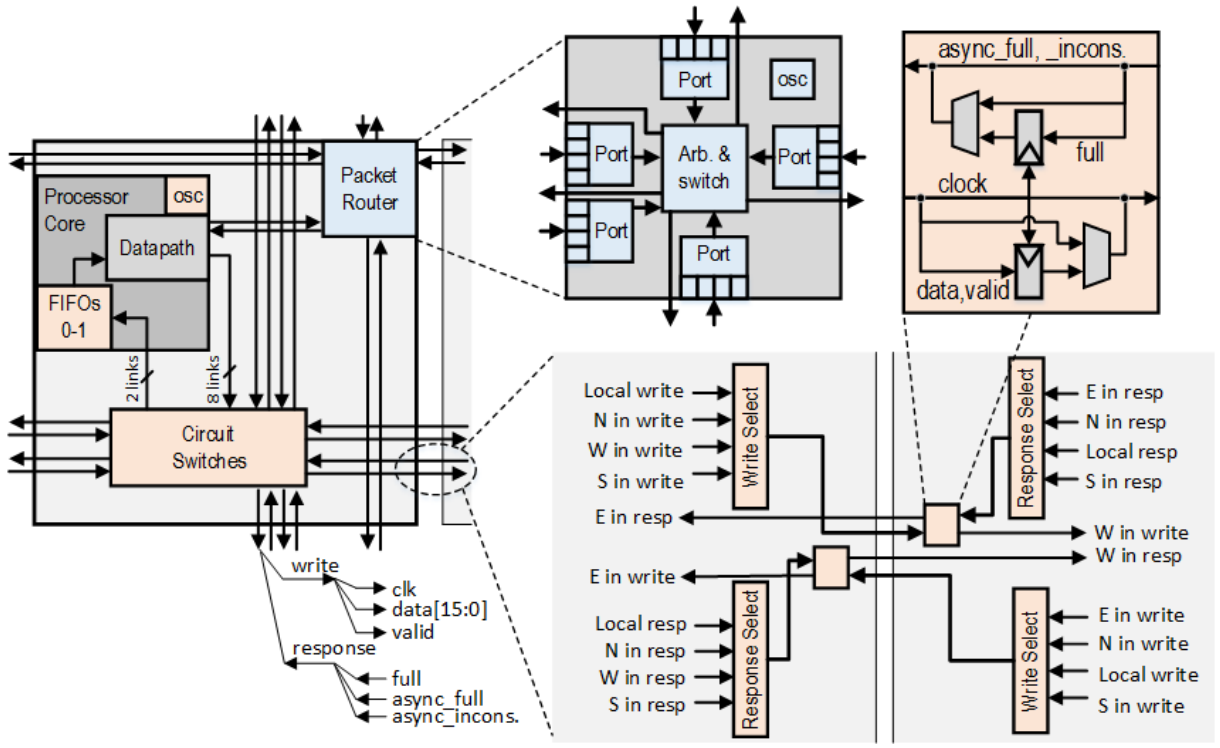


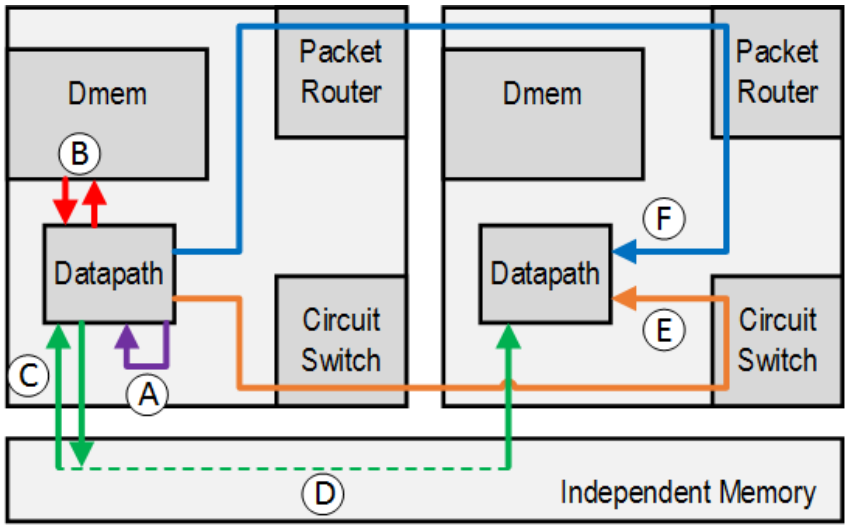
Figure 4.3: Overview of inter-core communication using circuit and packet networks. Writes are source-synchronous; responses include asynchronous wake-up signals for sleeping processors. Circuit links include configurable registers and an east-west connection for one layer is expanded on the right.

The circuit-switched links are source-synchronous so the source clock travels with the data to the destination, where it is translated to the destination-processor's clock domain. The network supports communication between adjacent and distant processors, as resources allow, with each link supporting a maximum rate of 28.5 Gbits/sec with optionally-inserted registers to maintain data integrity over long distances. Each of the four edges of each processor has two such links entering and two links exiting the processor. The high-throughput circuit-switched network is especially efficient—transferring data to an adjacent processor dissipates 59% less energy than writing and later reading that data using local data memory, and transferring that data to a processor 4 tiles away requires only 1% more energy than using data memory.

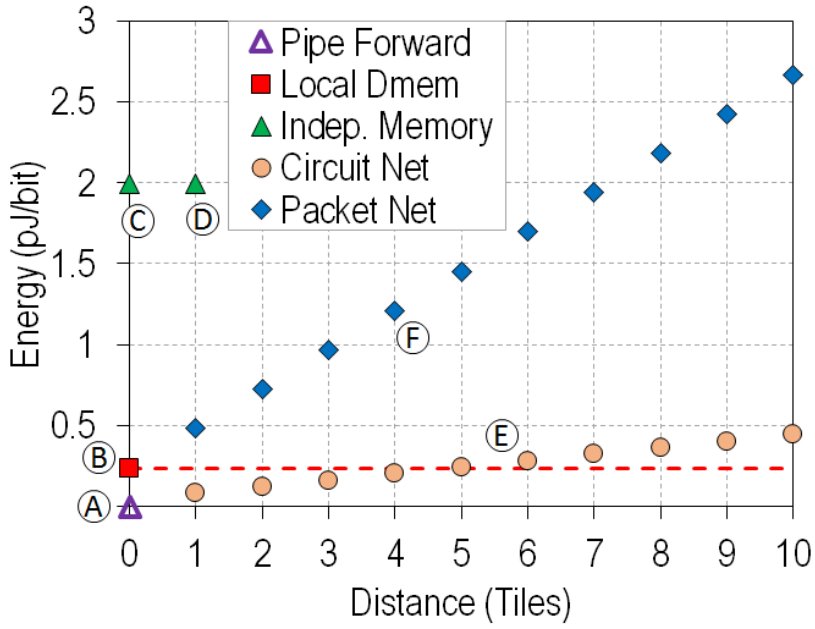
The packet router inside each processor occupies only 9% of each processor's area and is especially effective for high fan-in and high fan-out communication, as well as for administrative messaging. Each router supports 45.5 Gbits/sec of throughput with a maximum of 9.1 Gbits/sec per port. Routers operate autonomously from their host processors and contain their own clock oscillators so they can power down to zero active power when there are no packets to process. Each router contains five 4 x 18-bit input buffers, one for each cardinal direction and one for the local processor. Routers utilize wormhole routing to efficiently transfer long data bursts, in which a header packet will reserve a path and is followed by an arbitrary number of data packets, terminating in a tail packet which releases the path.

Each circuit or packet link terminates in a dual-clock FIFO memory [31] which reliably transfers data between clock domains. Additionally, links contain the necessary asynchronous wake-up signals which inform idle modules when they need to activate their local clock to process new work or to verify when FIFOs are full or empty. Both network types contribute to a total bisection bandwidth of 4.2 Tbits/sec.

Figure 4.4(a) illustrates the various methods of transferring data from one point in an application to another. These points may be within a single processor or spread across different processors depending on how code has been partitioned. Figure 4.4(b) reports the energy costs for each method and includes both a write and a single read, implying transferred data are used only once. Since pipeline forwarding is the lowest-energy method, energy values are reported as additional energy required beyond forwarding, that is, pipeline forwarding = 0.0 in this graph.



(a)



(b)

Figure 4.4: Path diagram and measured energies to transfer a bit of data from one point in an application to another versus distance, beyond the energy required for pipeline forwarding (i.e., pipeline forwarding = 0.0). (A) Pipeline forwarding or (B) local Dmem may be used for in-core transfers. Independent memory may be used for (C) local or (D) neighbor-processor transfers. Both (E) circuit and (F) packet networks support distant transfers.

4.1.3 Processor Data Memory Organization

Processors with a relatively small amount of memory per core require that memory is used efficiently. A straightforward solution to sustain a throughput of one instruction per cycle with common 2-input-operand and 1-output-operand instructions would be to utilize an N-word 3-port data memory which is unfortunately not very area or power efficient. If a 3-port memory is unavailable, one can be made easily albeit very inefficiently, from two N-word memories with write ports shorted together and reads made independently [4, 5]. A third possibility is to utilize two independent N-word memories which has the great advantage of yielding a total data space of 2N words and being able to sustain two reads and one write per cycle, but only if there are no conflicts where both input operands are in one of the two banks. Conflicts can be resolved by detecting their occurrence and stalling the processor when they occur. We have chosen another approach which uses compile-time information to place data into banks to minimize conflicts. When conflicts cannot be avoided or ruled out, data is written into both banks eliminating the conflict for that instruction and allowing a sustained throughput of one instruction per cycle at a cost of the loss of one otherwise-useful data word. Profiles of five diverse applications (AES encryption, 4095-bit code length low-density parity-check (LDPC) decoder, 100-byte database record sorting, 802.11a/g OFDM Wi-Fi receiver, and software single-precision floating-point arithmetic) showed that 99.66% of all operands across all applications could be mapped to an address in only one bank and thus only a very small number of operands needed to be written to both banks redundantly to avoid conflicts during subsequent reads. The scheme permits conflict-free addressing with optimal memory space maximization. Figure 4.5 shows the lightweight circuitry used to implement the three types of writes (bank0, bank1, both banks) and properly route read data to the processor.

4.1.4 Independent Memory Modules

Independent memory modules each contain a 64 kB SRAM and are shared between two neighboring processors. Modules support random access and a variety of programmable burst access patterns for data reading and writing, and are also capable of streaming instructions for large-program execution to an adjoining processor using an internal control module. When executing an instruction stream from an independent memory, a processor transfers program control and branch prediction control to dedicated circuits inside the memory block to more efficiently execute across branches. Programs running out of the external memory may use up

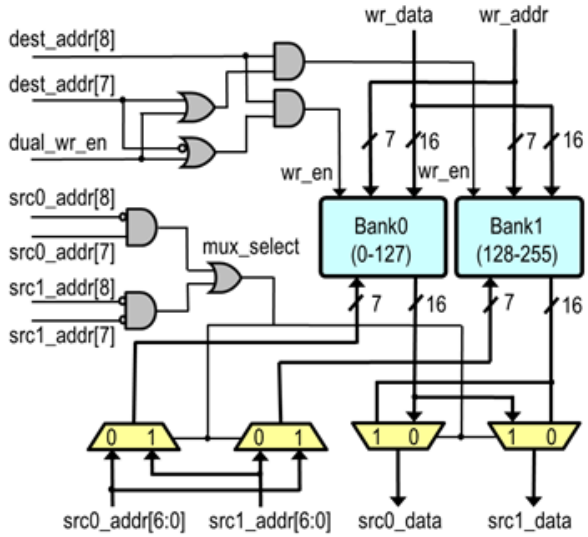


Figure 4.5: Multibank data memory read and write circuitry.

to 11,050 instructions. Each memory module contains two 32 x 18-bit input buffers, two 32 x 16-bit output buffers, and one 16 x 2-bit processor response buffer, and supports 28.4 Gbps of I/O bandwidth. Figure 4.6 gives details of the module's internal blocks.

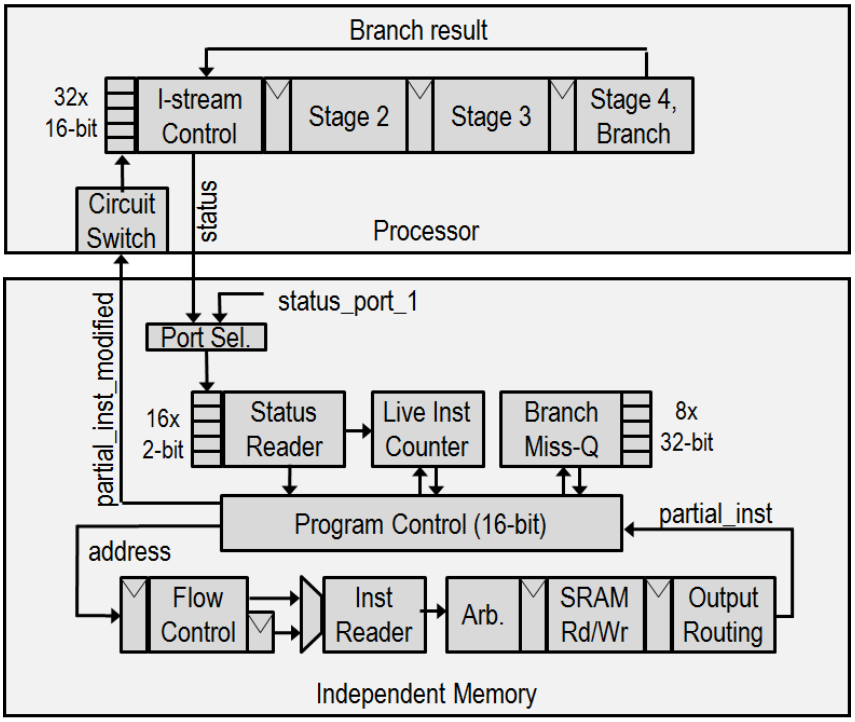


Figure 4.6: Components used in streaming instructions from a shared memory to a neighboring processor. Streaming logic is shared between two processors, with only the port 0 connection shown here.

4.2 Fine-grain Clocking

Many-core applications often require processors to remain idle or operate at low activity for substantial periods of time. Therefore, energy-efficient many-core designs must adapt to wide variations in core workloads. In KiloCore, each core, each packet router inside each core, and each independent memory module contains its own local programmable clock oscillator in an independent fully-synchronous clock domain [32] resulting in a total of 2012 Globally Asynchronous Locally Synchronous (GALS) [33] clock domains.

Oscillators do not use PLLs and each one is allowed to change its frequency, halt, or restart arbitrarily including with respect to other clock domains. Halting is very helpful in saving energy when there is no work to do which is detected by the processor when it attempts to read input buffer(s) which are empty or when attempting to write output buffer(s) which are full. Oscillator halting is handled automatically by local hardware logic which observes instruction source and destination operands, and also the state of inter-processor buffers for both upstream inputs and downstream outputs. When an oscillator is halted, the core/router/memory consumes zero active power. A halted processor consumes only 1.1% of its typical active power through leakage. Oscillators restart in response to asynchronous signals from connected cores when they send data to an empty buffer or free room in a full buffer, for upstream and downstream links respectively. Cores exiting a halt state require up to 3 cycles to read input buffers before program execution may continue; cores entering a halt state require a variable number of cycles after program execution pauses to complete any pending writes to the communication network. This inefficiency is negligible in many cases however it can be significant in situations where high-workload and low-workload cores are connected and performing fine grain communication, such that the low-workload core is regularly waiting on the high workload core, but not for long enough periods of time to benefit from clock halting. Per-core oscillator frequency tuning is used to help in this situation, slowing the low-workload core such that its data production or consumption rate is matched to the high-workload core. We estimate the ideal clock frequency for each core to be the lowest frequency at which a core may operate without reducing overall application throughput. These frequencies are identified during application profiling. Fortunately, even significantly inaccurate frequency estimates typically result in small increases in power dissipation over the ideal case. Tuning could certainly also be done during a run-time tuning phase or even during program execution by a dedicated hardware controller [4].

For the four applications described in Section VI, on average, clock halting yields a 61% reduction in energy usage compared to processors which never halt and do not utilize per-core frequency tuning. Of the energy that is consumed, 87% is from program computation and leakage while 13% is from stall cycles.

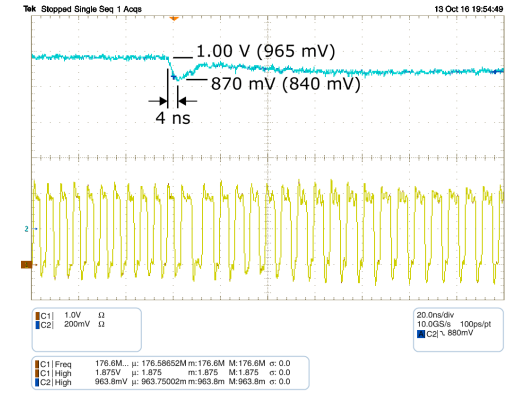
4.2.1 Tolerating Power Grid Voltage Variations

One thousand cores arbitrarily switching between being halted with leakage only to fully active can clearly result in significant power grid noise. Rather than trying to minimize the noise, clock oscillators are designed to rapidly adjust their instantaneous frequencies to compensate for supply noise variations through circuit design and by being powered by the local core's power grid.

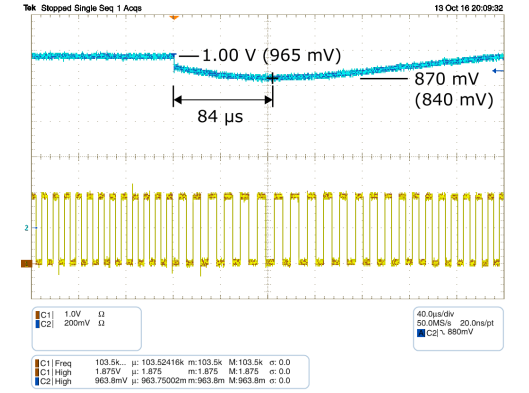
Oscillators are designed so that their frequency tracks closely below the core's maximum operating frequency when voltage droop occurs and in fact, cores were found to operate error-free when configured to operate at their maximum frequency without any additional margin for voltage droop, though some margin may be needed for overshoot depending on the power supply characteristics. In a manner similar to oscillators, circuit elements in the clock trees such as buffers and clock gates naturally adjust their instantaneously delay because they too are powered by the core's local power grid.

Figure 4.7 shows measured waveforms from a beyond-worst-case voltage droop event, where 999 processors are simultaneously turned on. In actual usage, only approximately 2/3 of the array could start at one time because halted cores are restarted by the arrival of external data sent from non-halted cores, and processors with outputs connected to more than two other processors at a time is very rare in our explored applications. In this test, a single victim processor in the center of the array at coordinates (15,15) runs a critical path test program while the other 999 cores in the array are simultaneously turned on to maximum frequency and begin running an energy intensive program. A globally-broadcast configuration signal is used to synchronize this event. The test is performed at a nominal supply voltage of 1.0 V. The victim processor was found to operate error-free throughout the event at a nominal clock frequency equal to its measured standalone maximum frequency to within 20 MHz, the oscillator step size at the test voltage.

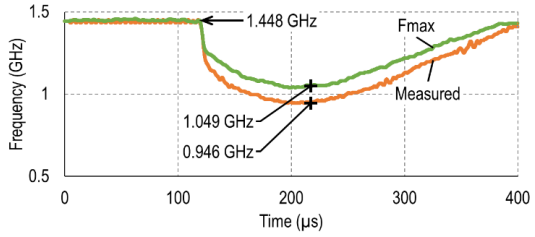
A measured on-die waveform of the supply voltage is shown in Figure 4.7(a), showing the short term supply noise using a 200 ns capture window. A 14% reduction in voltage occurs over 4 ns, with a corresponding decrease in the victim processor's clock period to compensate.



(a)



(b)



(c)

Figure 4.7: Supply voltage noise at a nominal 1.0 V when simultaneously turning on 999 processors from fully halted to fully active at maximum frequency, while measuring the clock oscillator of a single victim core in the center of the array. 1.0 V appears as 965 mV and 870 mV appears as 840 mV due to a resistor-divider created by our 1.9 Ohm SMA cable and 50 Ohm scope input. (a) 20 ns/division waveform capture, showing a clock frequency reduction in response to a 14% supply voltage reduction over 4 ns. (b) 40 us/division waveform capture, showing gradual supply droop and recovery, with a corresponding clock frequency recovery. (c) (Lower orange waveform) Instantaneous clock frequency calculated from the time-domain waveform in subplot (b), and (Upper green waveform) estimated Fmax derived from independent measurements, showing victim processor operation below but a maximum of 10% from its maximum possible operating frequency Fmax.

Figure 4.7(b) shows the long term voltage droop and recovery using a 400 µs capture window. 84 µs after the turn-on event, the voltage begins recovering from 13% below nominal as a function of the PCB and bench power supply electrical environment. To maintain visibility of the clock waveforms, the processor output clock is divided by 8 for subplot (a) and is reduced to a 630 MHz source frequency and divided by 8192 for (b). Figure 4.7(c) plots the victim processor's instantaneous frequency (labeled "Measured") during the test using a higher clock rate data capture, and with the same timescale as subplot (b). Also plotted is the processor's maximum supported frequency corresponding to the instantaneous voltage (labeled "Fmax"), based on pre-test measurements. The victim processor's oscillator remains below but within 10% of this maximum Fmax frequency. 84 µs after the turn-on event, the maximum frequency is reduced to 28% below nominal, while the oscillator's frequency is reduced 35%.

4.3 Design and Implementation

The processor array is built from standard cells and was synthesized except for small circuits such as the clock oscillator which were designed by hand. Cell placement and routing were performed by industry-standard CAD tools. Except for the 64 KB SRAMs inside the independent memory modules, all memories are built from clock-gated flip-flops with synthesized interfacing logic which greatly simplifies the physical design and lowers the minimum operating voltage for applications which do not use the independent memories.

The 8.0 mm by 8.0 mm chip was fabricated in a 32 nm partially depleted silicon on insulator (PD-SOI) technology and contains 621 million transistors. The entire array measures 7.94 mm by 7.82 mm. Each processor contains 575,000 transistors and occupies 239 µm by 232 µm; therefore 18 processors occupy almost 1 mm^2. Figure 4.8 is a die photo showing outlines of the 1000 cores and 12 independent memories, and 564 C4 solder bumps for flip-chip mounting in the center of the array. The chip is mounted inside a stock 676-ball BGA package that delivers full power to only the approximately 160 central processors; therefore, a maximum execution rate of 1.78 trillion MIMD instructions per second per chip is possible only with a custom-designed chip package. I/O signaling is handled by 64 LVDS drivers and 38 single-ended drivers. Pad drivers are placed along the periphery of the processor array. Ten analog voltage probe points are included to support on-chip voltage measurements.

Figure 4.9(a) is a post-placement plot of a single processor tile showing regions of the largest components along with details on the die area occupied by the various components. Both the

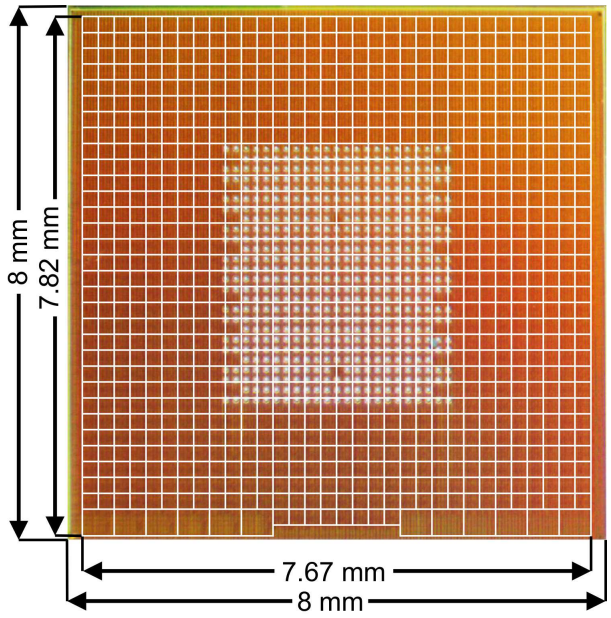
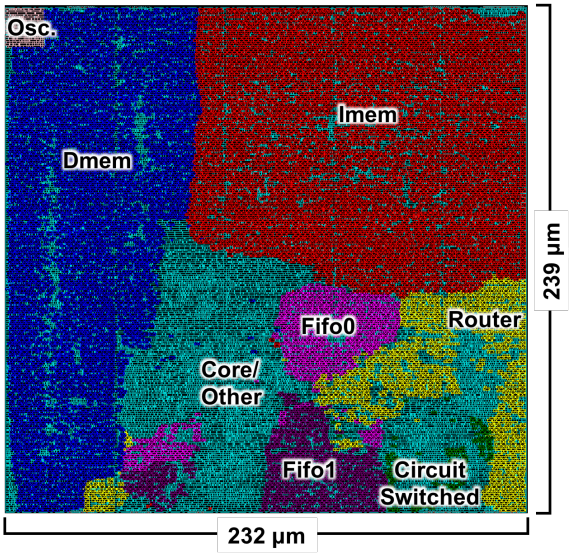


Figure 4.8: Die micrograph.

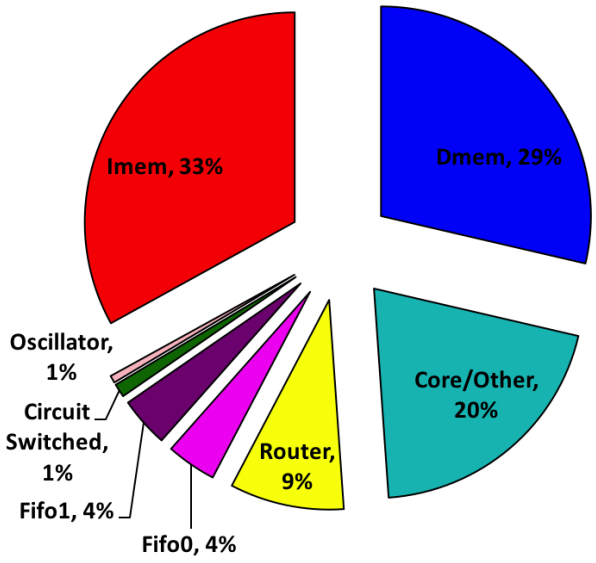
circuit-switched network (including FIFO0 and FIFO1) and the packet-switched network (includes router clock oscillator) occupy 9% of each tile's area. The processor's two clock oscillators and associated control occupy 1% of the tile's area and recover some of that area by eliminating the need for a chip-level clock tree. Figure 4.10 shows the same information for a single independent memory tile.

4.4 Measured Results

Processors, routers, and independent memories operate from a maximum voltage of 1.1 V down to minimum voltages of 560 mV, 670 mV, and 760 mV respectively. Figure 4.11 shows the average maximum frequency for each of these modules across their operable ranges. Independent memories have a reduced operating voltage due to their large SRAM array. The reason for the routers' reduced operating voltage is not known but suspected to be due to a specific implementation feature in their GALS network interfaces. Individual cores are allowed to operate at their local Fmax due to GALS clocking. At their highest voltage, processors average 1.78 GHz. When certain critical paths related to ALU carry and zero flags are avoided or coded with two instructions, a processor may operate up to 22% above its normal maximum frequency—a typical processor using this technique was measured operating at 2.29 GHz. This is done by a simple



(a)



(b)

Figure 4.9: (a) Annotated layout and (b) area breakdown of a single processor tile.

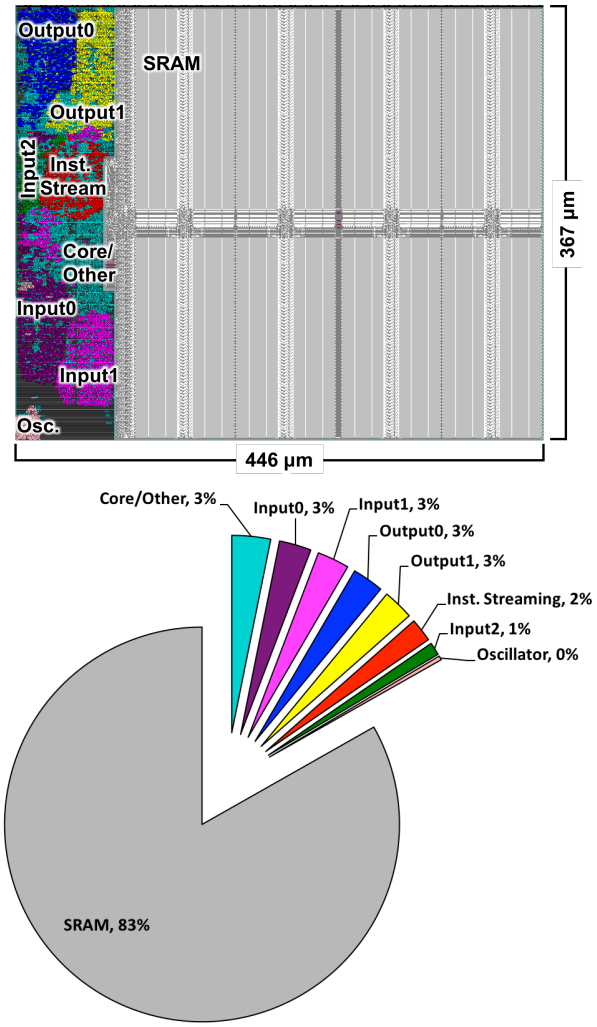


Figure 4.10: (a) Annotated layout and (b) area breakdown of a single independent memory tile.

reprogramming of the clock oscillator so that its frequency is appropriately higher than normal based on the critical paths used by the program assigned to that processor.

Table 4.1 lists energy usage of a variety of instructions and events when operating at 900 mV. ALU and MAC instructions are categorized according to their pipeline groupings, where input latches for each group isolate them from each other. Measurements are taken with high operand bit activity. Branch misprediction energy includes three high-activity instructions on the mispredicted path.

Figure 4.12 shows the typical energy per operation for each module type across its operable voltage range. Processor power varies considerably with instruction selection and memory access patterns. Therefore, processor instruction energy is calculated using weighted averages based on

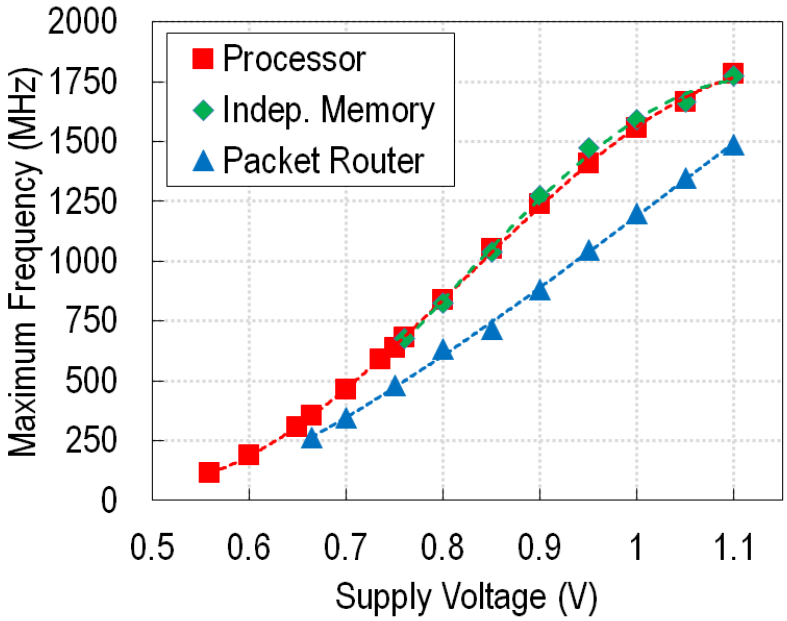


Figure 4.11: Maximum operating frequency of processors, memories, and routers.

Table 4.1: Energy per operation or activity at a supply voltage of 900 mV. Router flit transfer does not include clock energy; processor and memory operations include clock energy.

Operation or Activity	Energy (pJ)
Instruction, ALU Add/Sub	11.0
Instruction, ALU Logic	10.3
Instruction, ALU Move	10.0
Instruction, ALU Shift	9.9
Instruction, ALU Other	9.7
Instruction, MAC	19.9
Instruction, Branch Correct	9.7
Instruction, Branch Incorrect	41.0
No-op	7.5
Stall Cycle	6.9
Dmem Read	1.0
Dmem Write 1 Bank	2.7
Dmem Write 2 Banks	5.0
Circuit Comm. First Tile	1.3
Circuit Comm. Additional Tiles	0.6
Packet Router Clock	2.2
Packet Router Flit Transfer	1.9
Shared Memory Stall Cycle	4.5
Shared Memory Read	12.3
Shared Memory Write	19.6

code from a profiled 334-processor FFT application, including data reads/writes along with circuit network communication. At 560 mV, a single processor dissipates 5.3 pJ per typical instruction while operating at 115 MHz. Packet router energy varies with port activity level; values reported are for transferring a single flit, assuming two router ports are actively transferring and sharing clock energy. Independent memory energy also depends on activity and so values reported are an average of random read and random write energies, and are further averaged between one or both ports active.

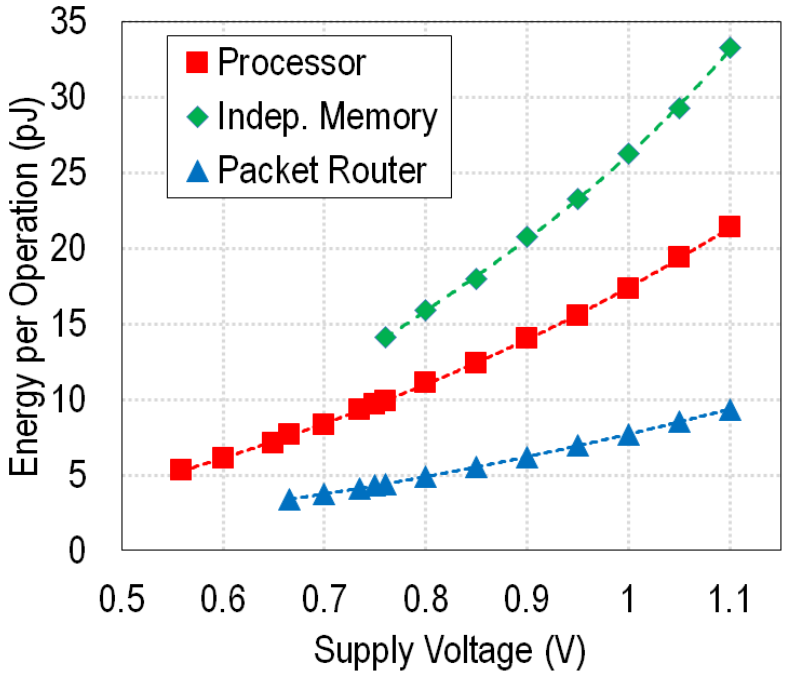


Figure 4.12: Energy per typical operation for processors, memories, and routers.

Figure 4.13 shows the power for each module across its voltage range when active 100% of the time utilizing the same weightings and conditions as were used for energy measurements.

4.5 Applications

Programming is accomplished by a multi-step process. Individual programs are written using C++ or assembly, making use of templating and parameterization to share code across processors. An automatic mapping tool maps tasks to cores with considerations such as: avoiding faulty or partially-functional processors; optimizations to take advantage of process, voltage, and temperature variations; self-healing for failures due to wear-out effects; and simultaneous execution of unrelated workloads.

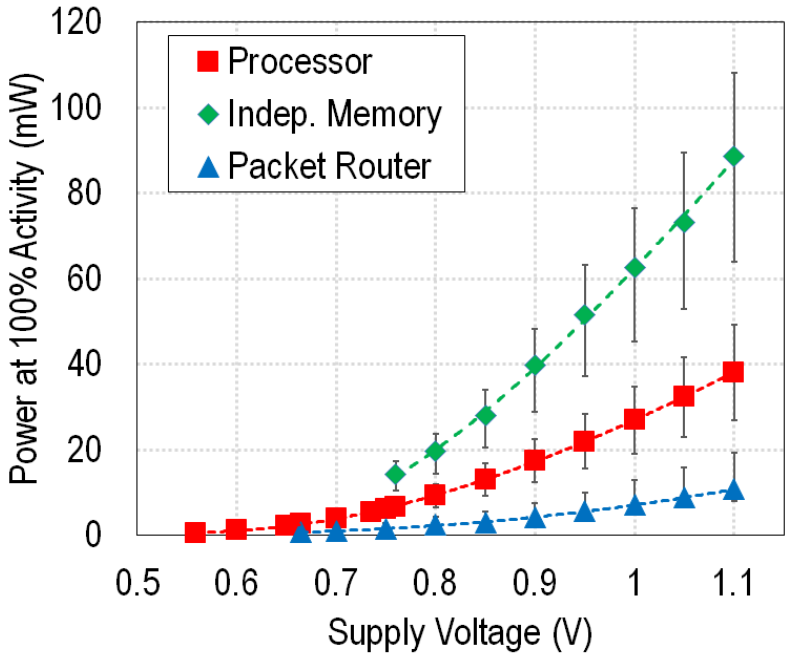


Figure 4.13: Power of a processor, memory, and router when 100% active and operating at the maximum clock frequency at the indicated supply voltage. Type of activity impacts power usage; the spread between low-energy and high-energy activities are indicated.

Several applications have been mapped to KiloCore and their performance estimated using simulations which assume custom chip packaging, whereby the cores along the outer part of the array are assumed to have the same voltage stability as those under the actual package's C4 solder bumps. Simulations are cycle accurate within a core, use sub-cycle precision for core interactions, fully model varied per-core frequencies, and utilize sub-instruction energy measurements. Application code has been lightly-to-moderately optimized and additional effort would yield significant improvements. All four applications store instructions inside local processor memories and so usage of independent memories, run-time instruction swapping, or run-time off-chip instruction streaming are not required.

An Advanced Encryption Engine (AES) application is implemented with 974 processors, expanding the 137-core version developed for AsAP2 [20]. At a reduced 0.9 V, it supports a throughput of 14.5 Gbps while using 6.5 Watts. It is operable down to 560 mV, where a throughput of 1.23 Gbps is achieved using 158 mW.

A Low Density Parity Check (LDPC) decoder is implemented with 944 processors and 12 independent memories, expanding the version discussed in chapter 2. It supports a (4095,3717)

code with row and column weights of 64 and 6, and utilizes 12 parallel decoding lanes. At a reduced 0.9 V, with four decoding iterations, it has a throughput of 111 Mbps while using 3.4 Watts. It is operable down to 760 mV, where it decodes 62 Mbps using 1.1 Watts.

A 4096-point complex Fast Fourier Transform (FFT) application is implemented with 980 processors and 12 independent memories, being freshly developed for this architecture. It processes 16-bit complex data and calculates 12 transforms in parallel. At 0.9 V, 567 MSamples per second are processed using 4.1 Watts. It is operable down to 760 mV, with 313 MSamples per second using 1.4 Watts. A second 4096-point complex FFT application was developed which processes a single FFT transform at a time and uses 619 processors and 12 memories to transform 295 MSamples per second using 2.6 Watts at 0.9 V.

The first phase of an "external" record sort is implemented with 1000 processors. This is based on the sorting application for AsAP2 [21], but is freshly implemented to better leverage KiloCore's architecture. 100-Byte records contain a 10-byte sorting key and are processed into sorted blocks of 185 kB in support of the second merging phase of the external sort. At 0.9 V, this application sorts 1.47 GB per second using 1.2 W. It is operable down to 560 mV, where it can sort 137 MB per second using 61 mW.

4.5.1 Task Partitioning

KiloCore is designed for high cooperation between processors, where each processor executes a small task of up to 128 instructions. Mapping an application to this architecture involves applying a series of task partitioning transformations, where the final tasks are mappable to the processors. These transforms are loosely categorized as serial and parallel partitioning.

Serial partitioning transforms sections of code into a sequence of tasks, which form a computation pipeline. Live variables at the code separation points are transferred between tasks using message passing. Variables may be transferred from producers to consumers either directly, through intermediate tasks in the chain, or using a mixture of these methods. Partitioning may produce tasks with as little as one instruction which directly reads data from the network, performs an operation, and writes the result back to the network.

Parallel partitioning performs task replication to increase the throughput of critical paths in the application which exhibit data parallelism. This transform is typically applied to loop bodies or is used to implement vector operations. This partitioning introduces overhead for splitting and joining the data being processed by the replicas, which may involve inserting additional

data routing tasks if a large number of replicas are formed. When task execution time varies significantly with the data, intelligent distribution may be used to supply data to tasks as they finish their computations.

Serial and parallel partitioning transformations are applied to the application multiple times, progressing from the original code to that which will be mapped to KiloCore. Figure 4.14 shows an example of partitioning a task which processes elements in a 4096 element data array. This task is partitioned serially to isolate the data access tasks from the main workload, replicating the loop iterator to maintain correct task execution counts. Parallel partitioning is then applied to accelerate the address generation and data computation tasks, with appropriate loop count modifications.

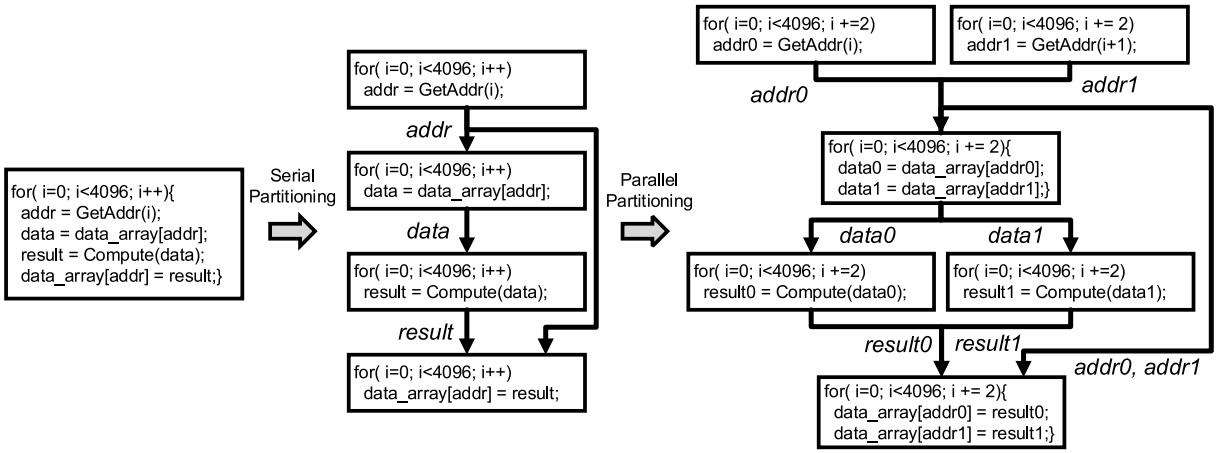


Figure 4.14: Example of serial and parallel task partitioning. Serial partitioning reduces instruction counts per task and isolates large data structures, while parallel partitioning improves the throughput of critical paths.

Figure 4.15 shows the number of instructions required for tasks after partitioning was performed for the sampled applications. All tasks fit within the 128-word instruction memory of a KiloCore processor.

Figure 4.16 shows the amount of data memory required by these same tasks. 98.7% of tasks fit within the 512-byte data memory of a KiloCore processor. The remaining tasks include those which access larger data structures in the FFT and LDPC applications. These tasks are mapped to the 24 processors neighboring the 12 independent memories in KiloCore.

Task partitioning introduces overhead for inter-task data transfers. This overhead is partially hidden in KiloCore by allowing instructions to directly access the network links as part of their source and destination fields. In the sampled applications, after partitioning, communication

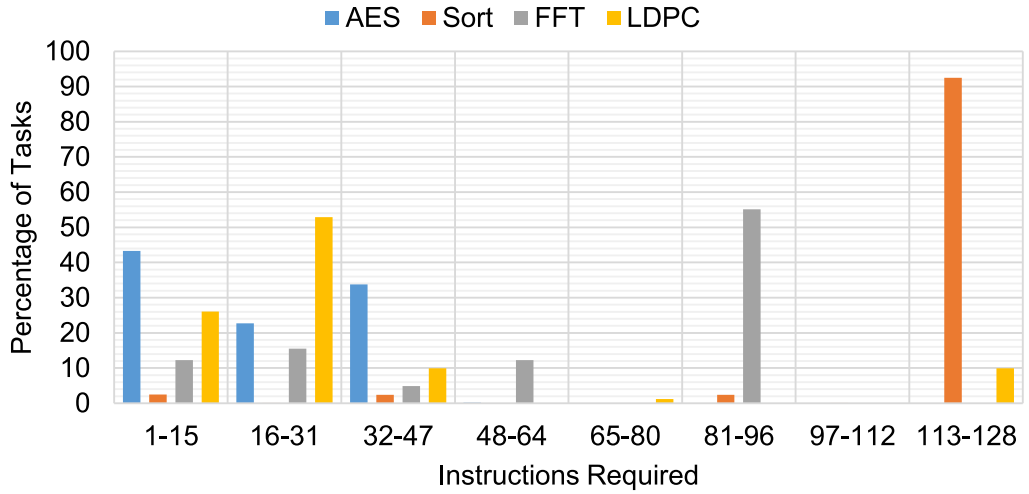


Figure 4.15: Number of instructions required by tasks in the example applications after task partitioning. All tasks fit within the 128-word instruction memory of a single processor.

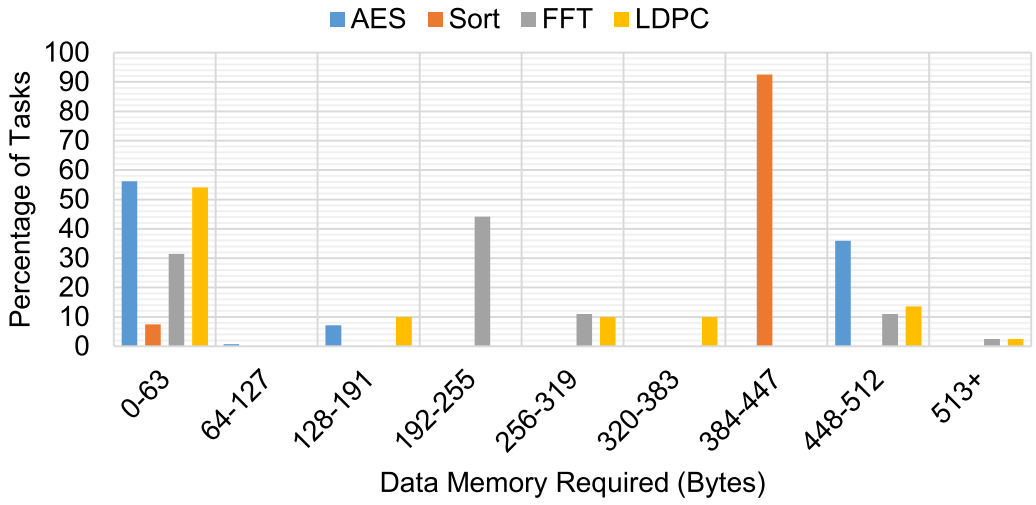


Figure 4.16: Amount of data memory required by tasks in the example applications, after task partitioning. Most tasks fit within the 512-byte data memory of a single processor, with a small number of tasks requiring the assistance of the independent memory modules.

overhead accounts for 30% of overall energy usage, including network energy along with instructions dedicated to reading or writing the networks. This energy partially replaces that which would be spent on writing and subsequently reading variables from local data memories. In some situations, partitioning will directly lower application energy, as a variable transferred using the circuit network over a short distance requires as little as 25% of the energy used when storing and reading the same variable from a local data memory.

Figure 4.17(a) shows throughput scaling for the sampled applications as core count increases. Figure 4.17(b) shows the corresponding energy efficiency of the applications. AES, FFT, and LDPC show approximately linear growth in throughput with core count, with energy efficiency remaining steady when going to large numbers of cores. The Sort algorithm is omitted here since it utilizes additional cores to increase the size of sorted blocks, and the amount of work being done at different core counts is not directly comparable.

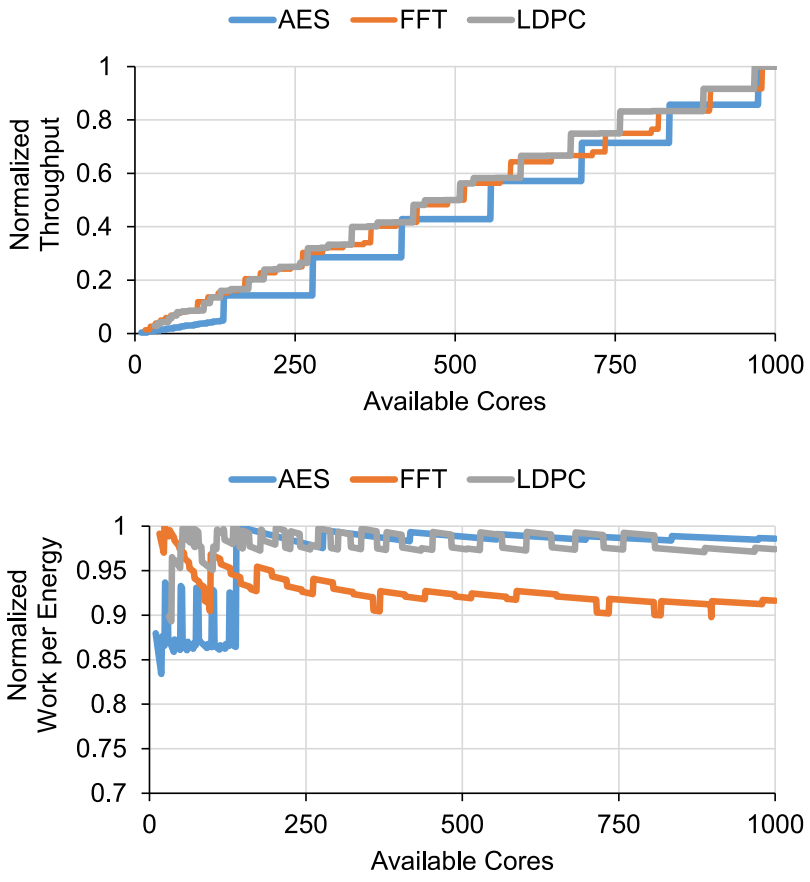


Figure 4.17: Normalized application (a) throughput and (b) energy efficiency as the number of cores available to the application is increased to 1000.

4.5.2 Task Networking

An important consideration for manycore architectures is how to transfer data between cores in a manner that is energy efficient, avoids network congestion, and supports inter-task synchronization. The communication requirements of an application depend heavily on the method of implementation. In our sampled applications, algorithms were chosen which, once partitioned into fine-grained tasks, exhibit low densities of inter-task communication links. When mapping one task to each processor, on average only 0.15% of possible links are utilized as compared to maximally dense all-to-all communication. In Figure 4.18, tasks are categorized by their number of required input or output connections. 95% of the tasks have a fan-in of two or fewer, and 93% have a fan-out of one or two. This result is partially influenced by the partitioning algorithms used, which favor reducing the number of links needed.

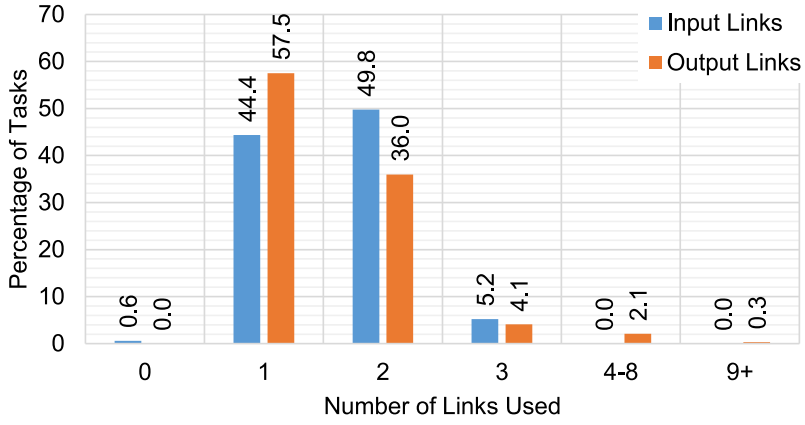


Figure 4.18: Tasks of sampled applications categorized by their number of inter-task input and output links. A large majority of tasks utilize less than 3 inputs and less than 3 outputs.

KiloCore utilizes complementary circuit and packet networks to efficiently support these links. The low area, low energy, high throughput circuit network supports 2 inputs and up to 8 outputs per processor. 95% of links in the sampled applications are supported by this network. The remaining links are assigned to the packet network, which is designed for medium throughput to reduce packet router area overhead. The packet network is also used to support general administrative signaling.

4.6 Performance and Comparisons

Performance metrics at a supply voltage of 1.1 V for applications are found in Table 4.2, including branch correct prediction rates, instructions per cycle and per second, and overall throughput and power.

Table 4.2: KiloCore application metrics for operation at 1.1 V. *Does not include time spent waiting for networks.

Application	Cores	Branch Correct Predict	Active Inst. Per Cycle Per Core*	Inst. Per Second (Billion insts/s)	Throughput	Power (W)	Throughput / Watt
AES	974	100%	.93	802	21.4 Gbps	15.4	1.39 Gbps/W
LDPC 4095-bit 5-iterations	944	89%	.95	320	145.4 Mbps	8.21	17.7 Mbps/W
FFT 4096-point complex	980	94%	.98	407	823.9 MSamp/s	9.75	84.5 MSamp/s/W
Sort 100-Byte records	1000	96%	.96	131	2.12 GB/s	3.12	0.678 GB/s/W

KiloCore's applications are compared against a selection of Intel i7 and Nvidia GPU processors due to their wide acceptance and deployment, highly-optimized hardware, and mature programming tools. In addition, Intel Core (and related Xeon) and Nvidia GPUs are frequently deployed in computing domains ranging from mobile, desktop, server, datacenter, to scientific supercomputer. Comparison data are given in Table 4.3 and include KiloCore data both unscaled and scaled to the same technology using data from Holt [8]. The AES comparisons on an Intel i7 [34] and Nvidia GPU [35] are taken from the literature, and do not use the specialized AES hardware present in many Intel processors. The LDPC comparisons on an i7 [36] and GPU [37] implement (9216,4608) and (2304,1152) codes with row and column weights of 6,3 and 24,12 respectively, and perform 5 decoding iterations. The i7 FFT uses the FFTW library with 8 independent threads iterating on cached data. The GPU FFT is implemented on an Nvidia GTX 960 using the cufftExecC2C function from the Nvidia Cuda cuFFT library. Both implementations utilize single precision floating point operations. Interestingly, a hypothetical floating-point KiloCore would actually experience a speedup compared to this fixed-point version which must explicitly handle data alignment and overflow functions. Sorting is implemented on an i7-3770k using std::sort in C++ with 8 independent threads operating on separate record groups in cache, and is implemented on a GTX 960 using the sort function from the Nvidia Cuda Thrust library. Power is measured using on-die energy counters when available, or by the measured power delta with a correction for power supply efficiency. For cited designs without reported power, we use half of the thermal design power (TDP) [38]; i7 power numbers do not

include uncore power. Area comparisons are made using die area, subtracting the estimated area for the graphics, memory controller, and unused cores in the comparison CPUs.

Table 4.3: Application metrics and comparisons of KiloCore with CPU and GPU implementations. KiloCore metrics are normalized against the comparison device, and are (cols. 7-8) unscaled and operating at 1.1 V, and †(cols. 9-10) scaled to the same technology using data from Holt [8]. ‡Assumes device power is half of thermal design power [38].

App	Device	Tech (nm)	Thruput	Thruput/ Watt	Thruput/ Area	KiloCore Relative Thruput/ Area	KiloCore Relative Thruput/ Watt	KiloCore† Relative Thruput/ Area	KiloCore† Relative Thruput/ Watt
AES	i7 920 [34]‡	45	1.2 Gbps	0.018 Gbps/W	0.11 Gbps/mm2	55.7	75.3	24.1	52.8
AES	Tesla C2050 [35]‡	40	60 Gbps	0.50 Gbps/W	4.6 Mbps/mm2	2.95	2.76	1.68	2.07
LDPC	i7 3960X [36]‡	32	180 Mbps	2.77 Mbps/W	1.11 Mbps/mm2	4.03	6.40	4.03	6.40
LDPC	GTX Titan [37]‡	28	621.4 Mbps	4.97 Mbps/W	0.41 Mbps/mm2	2.05	3.56	2.73	3.97
FFT	i7 3770k	22	1048 MSamp/s	18.4 MSamp/s/W	22.5 MSamp/s/ mm2	1.17	4.58	2.71	6.65
FFT	GTX 960	28	5120 MSamp/s	76.4 MSamp/s/W	6.55 MSamp/s/ mm2	0.57	1.11	0.76	1.23
Sort	i7 3770k	22	3.91 GB/s	0.074 GB/s/W	0.59 MB/s/mm2	0.80	9.11	1.87	13.2
Sort	GTX 960	28	0.135 GB/s	0.004 GB/s/W	0.024 GB/s/mm2	55.7	186	74.1	207

Across these applications and when scaled to the same fabrication technology, KiloCore at 1.1 V has geometric mean improvements of 4.3x higher throughput per area and 9.4x higher energy efficiency compared to the other processors. Significantly higher efficiencies are possible at lower supply voltages. At KiloCore's optimal energy-delay voltage of 0.9 V, it achieves geometric mean improvements of 3.1x higher throughput per area and 16.7x higher energy efficiency, with an overall 70.5x improvement in throughput per area per Watt. When comparing to just CPUs or just GPUs, this overall metric is 68.9x and 72.0x better respectively.

4.7 Development History

The work that would eventually become KiloCore began as a proof-of-concept design. The register-transfer-level (RTL) code (in Verilog) for a processor core was developed from scratch,

reusing none of the code from AsAP2. In addition to implementing ideas discussed in chapter 3, this architecture made numerous smaller changes to improve hardware efficiency or enable later feature expansions. A new assembler was written in Python, and the record sorting application was adapted as the primary test. Approximately six weeks were spent on initial microarchitecture design and Verilog and Python code writing, followed by two days of debugging to go from the first Verilog compilation attempt to successfully running the multiple-core Sorting application.

This work was set aside for a period of time to work on other projects. When the chip fabrication opportunity arrived it was on short notice, only 87 days before what would be our tapeout. Approximately six weeks of this time was spent on completing and polishing the RTL design, while co-authors handled preparing the physical design flow. Full access to the physical design libraries were gained only 34 days before tapeout.

The success of KiloCore, despite the rushed schedule, can partly be attributed to the robustness of this many-core architectural style: the independence of each processor allows them to be replicated without overhead for global buses, global clock trees, or shared cache management. In the years since tapeout only three bugs have been discovered in the RTL design, each solved by minor coding restrictions.

Chapter 5

KiloCore2

KiloCore2 (KC2) consists of 697 efficient, programmable processors to run software programs, 697 packet routers that are each paired with a processor, 2 Viterbi accelerators, 1 FFT accelerator, and 14 memory modules containing 64kB of memory each that may be used for data or instructions. The core layout of this array is displayed in Figure 5.1.

Similar to the first KiloCore, each element has an independent oscillator for local clock generation, and communicates with asynchronous neighbors using dual-clock FIFOs. The communication network includes two independent, statically configurable circuit meshes that support source-synchronous communication, a packet network which support dynamic communication, and a dynamic circuit network which uses packets to establish temporary source-synchronous links during run-time.

5.1 Summary of major differences

A number of changes were made to the architecture since KiloCore. These changes were motivated by a mixture of simple refinements (pipeline changes), extra design time (new packet router), and a custom package design (multiple voltage domains, more I/O). A partial list of changes follows.

- New low-area packet router
- 3 voltage domains available to each processor, up from 1
- DVFS control circuitry
- Deepened multiplier pipeline and a 63% higher processor frequency target
- FFT and Viterbi accelerators

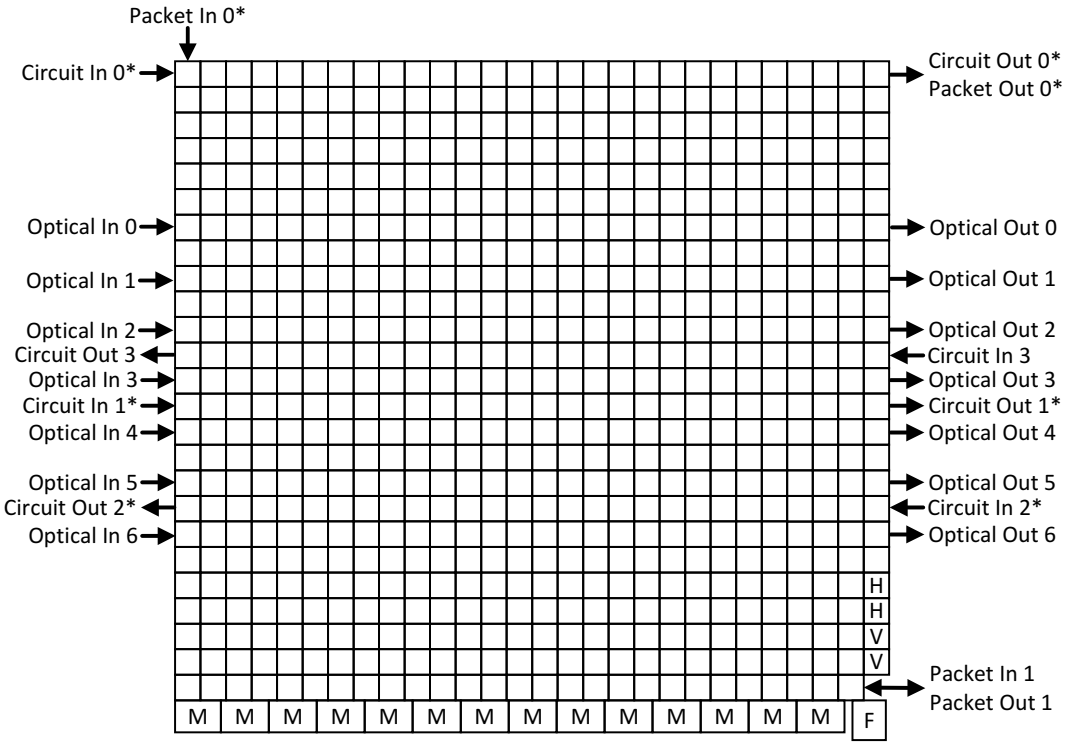


Figure 5.1: Layout and IO ports of KiloCore2, where an asterisk (*) denotes higher speed LVDS differential ports. "M" is a shared 64kB memory; "V" is a Viterbi accelerator; "F" is an FFT accelerator; "H" is a high-speed processor. The empty space above the FFT accelerator is occupied by a temperature-voltage sensor, which is slightly smaller than a processor.

- Temperature/voltage sensor
- Special high speed processors with an even deeper general pipeline and removal of some critical path functionality, having a 200% higher frequency target than original KiloCore processors
- Double data-rate chip IO, transferring on both clock edges
- Many more IO ports: 4x circuit network ports, 2x packet network ports, and 14 narrow ports intended for integration with external optical IO drivers, as shown in Figure 5.1
- Various small refinements: reduced opcode width, more branch options, byte level memory access, etc.

Table 5.1 shows the amount of RTL source code involved in each architecture, from AsAP2 through KiloCore2. KiloCore and AsAP2 share no code. KiloCore2 uses modified versions

of the AsAP2 FFT and Viterbi accelerators. The AsAP2 motion estimation accelerator has no comparable block in either KiloCore. Generated code files are omitted, such as the 375-thousand-line full-array connections file which is written by a configurable Python script for KiloCore.

Table 5.1: Lines in RTL Verilog code files, given as entire file and just code, from AsAP2 to KiloCore2, omitting generated files such as full array connections. AsAP2 and KiloCore share no code. KiloCore2 uses modified versions of the AsAP2 FFT and Viterbi accelerators. The AsAP2 motion estimation accelerator has no comparable block in either KiloCore.

Chip	Source Lines	Code Lines
AsAP2	47012	32184
KiloCore	31088	20885
KiloCore2	65940	44049

5.2 Specialized Cores

The bottom right of the array is occupied by specialized cores, as visible in Figure 5.1. These are described below.

5.2.1 High Speed Processor

High speed processors have reduced capabilities, in return for higher operational frequencies. Notable changes include: the removal of the MAC unit, tied off external memory port, disconnection from the packet input port, removal of network induced stall logic (in favor of polling), removal of dynamic voltage selection, removal of address generators, halved data memory size, removal of branch prediction, and an additional ALU execution stage. Programs written for high speed processors must handle data flow control in software. In return for these trade-offs, high speed processors are able to execute their kernels nearly twice as fast as standard processors, and excel at executing serial code.

5.2.2 Accelerators

The Fast Fourier Transform accelerator contains specialized hardware for performing FFTs up to length 4096, and inherits from the design used in AsAP2 [19]. The primary modification is a doubling of module IO ports, which were previously limiting FFT performance. For a 4096-point FFT, the AsAP2 design requires 8193 cycles for IO but only 6174 cycles for compute; with compute and IO being performed in parallel, this results in compute resources only being utilized 75% of the time. In KC2, 4097 cycles are required for IO, leading to a 33% speedup at this FFT length.

The Viterbi accelerators contain specialized hardware for performing decoding of Viterbi encoded data, and inherit from the design used in AsAP2 [4]. Wrapping logic for these accelerators integrates them into the architecture's network, clocking, and configuration schemes.

5.2.3 Temperature/Voltage Sensor

The TVsense module provides on-die temperature and voltage measurements. This module is slightly smaller than a processor, and is located in the lower-right portion of the array near the accelerators and high speed processors. Programming and reading are done through the standard array configuration and test bus.

5.3 Design and Implementation

Similar to the first KiloCore, the processor array is built from standard cells and is synthesized except for small circuits such as the clock oscillator which were designed by hand. Cell placement and routing are performed by industry-standard CAD tools. Except for the SRAMs inside the independent memory modules and FFT and Viterbi accelerators, all memories are built from clock-gated flip-flops with synthesized interfacing logic which greatly simplifies the physical design and lowers the minimum operating voltage for applications which do not use those modules.

The 8.0 mm by 8.0 mm chip was fabricated in a 32 nm partially depleted silicon on insulator (PD-SOI) technology and contains 580 million transistors. The entire array measures 7.51 mm by 7.43 mm. Each processor contains 750,601 transistors and occupies 265 µm by 274 µm. Figure 5.2 shows the top view of the array in Encounter, with top level power and ground rails hidden.

KiloCore2 contains 2,499 C4 solder bumps for connecting to a custom BGA chip package. Of these, 296 are used for double-ended (LVDS) and 161 for single-ended I/O signals. 12 bumps are set aside as analog probe points, to be used in sampling the power and ground voltages near selected processors. 2 bumps are utilized as dedicated power and ground for the temperature-voltage sensor. The remaining bumps are used for general power delivery: 81 for I/O, 531 for standard core voltage, 200 and 196 for alternate processor voltages, and 1020 for common ground. I/O signals and power are placed near the periphery, while core power is distributed throughout.

Standard processors and routers are designed to operate at 2.0 GHz at 900 mV. This achieves a 63% higher throughput per processor than the original KiloCore. Specialized high speed processors are designed to operate at 3.85 GHz at 900 mV. At 1.1 V, the standard and high

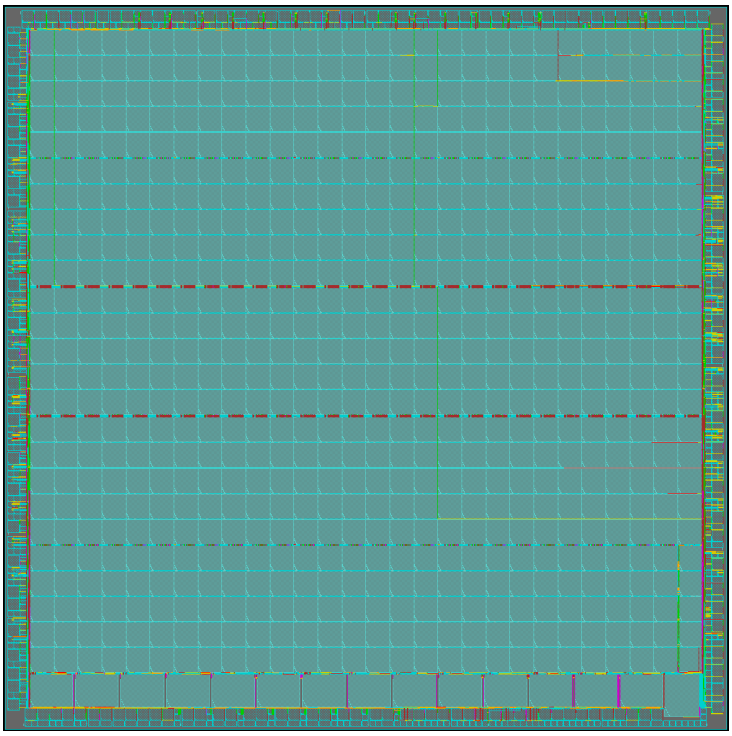


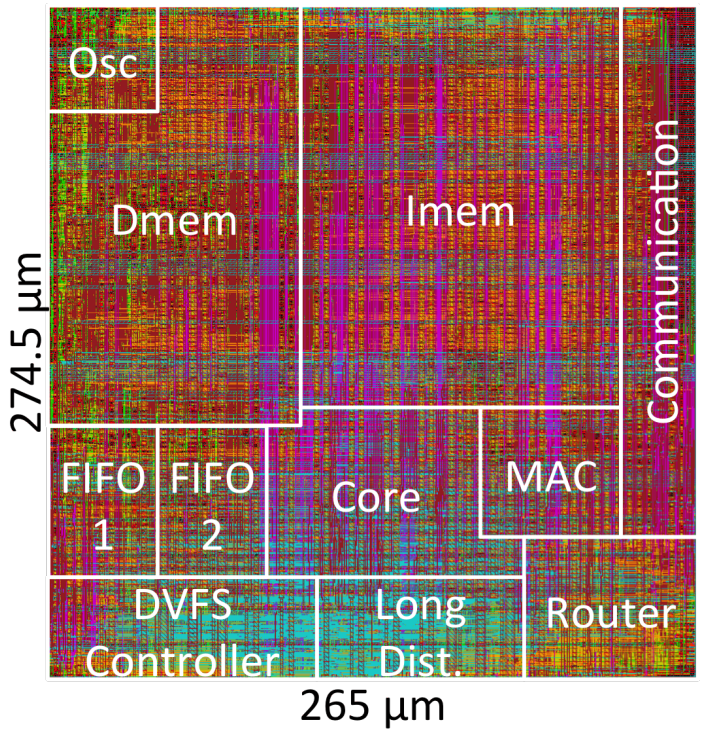
Figure 5.2: Top view of KiloCore2 in Encounter, when hiding power and ground wires. Layout corresponds to that shown in Figure 5.1. The periphery consists of off-chip I/O drivers, Electrostatic discharge triggers and clamps, and deep trench capacitors [39].

speed processors are projected to reach 2.9 GHz and over 5 GHz respectively. The entire array is projected to achieve over 2 tera-operations per second when running at 1.1 V.

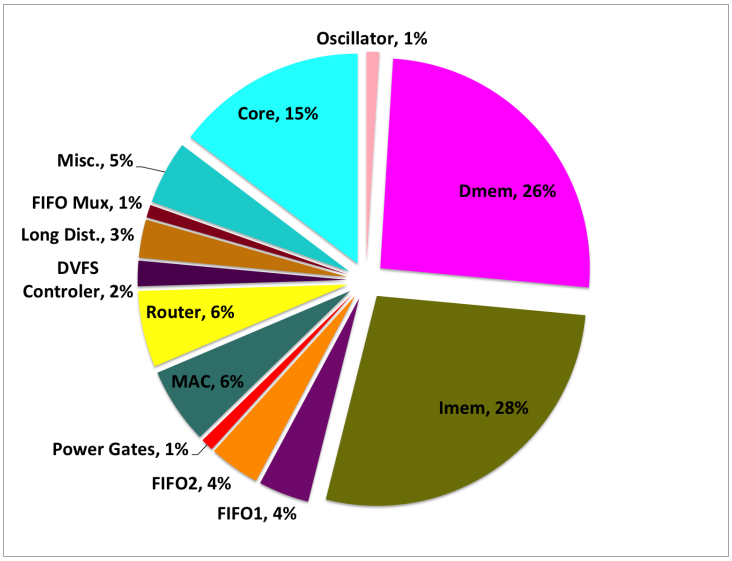
Figure 5.3(a) is a post-placement plot of a single processor tile showing regions of the largest components along with details on the die area occupied by the various components. The circuit-switched network (including two 64 Byte input FIFOs) occupies 11% of the tile area, whereas the new packet router occupies only 6%. The processor's clock oscillator and associated control occupy 1% of the tile's area and recover some of that area by eliminating the need for a chip-level clock tree.

Annotated layouts for other tile types are shown in Figure 5.4 for high speed processor tiles, Figure 5.5 for shared memory tiles, Figure 5.6 for FFT accelerator tiles, and Figure 5.7 for FFT accelerator tiles.

At the time of this writing, the KiloCore2 chips are pending final touch-ups and daughterboard mounting.



(a)



(b)

Figure 5.3: (a) Annotated layout and (b) area breakdown of a single KiloCore2 standard processor tile [39].

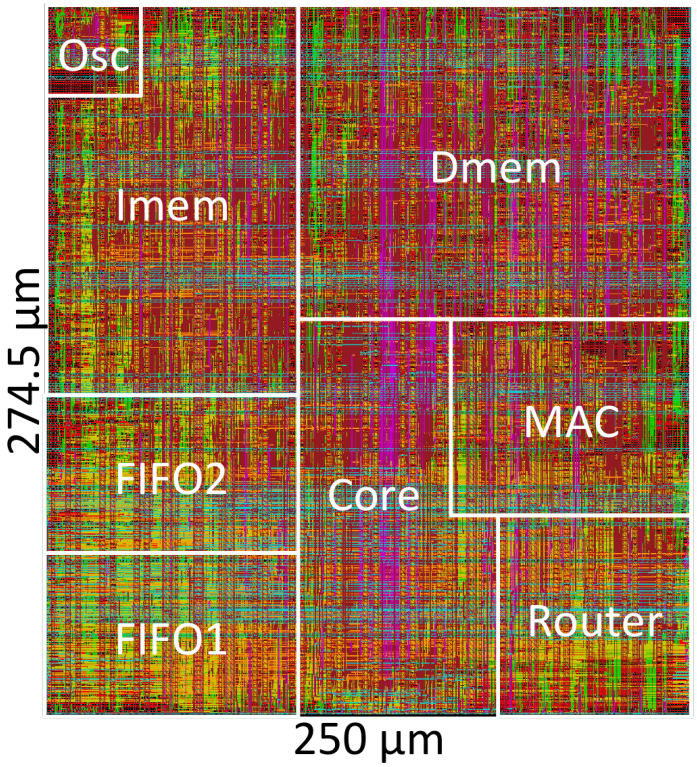


Figure 5.4: Annotated layout of a single KiloCore2 extra-high-frequency processor tile [39].

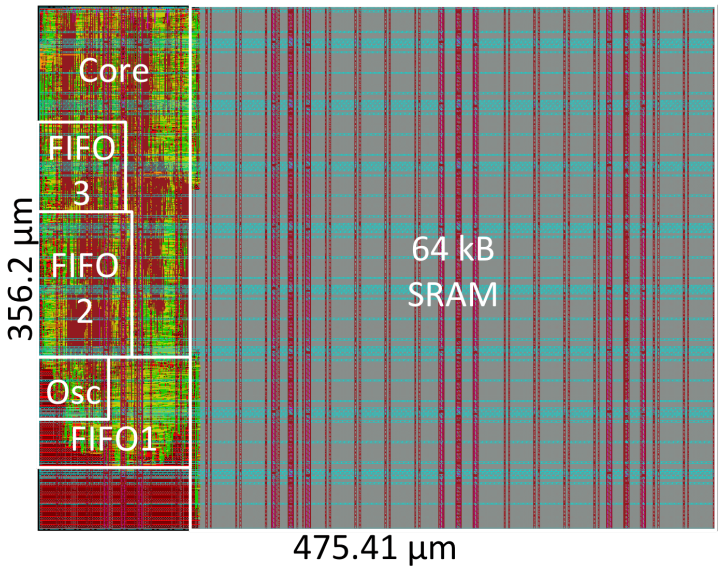


Figure 5.5: Annotated layout of a single KiloCore2 shared memory tile [39].

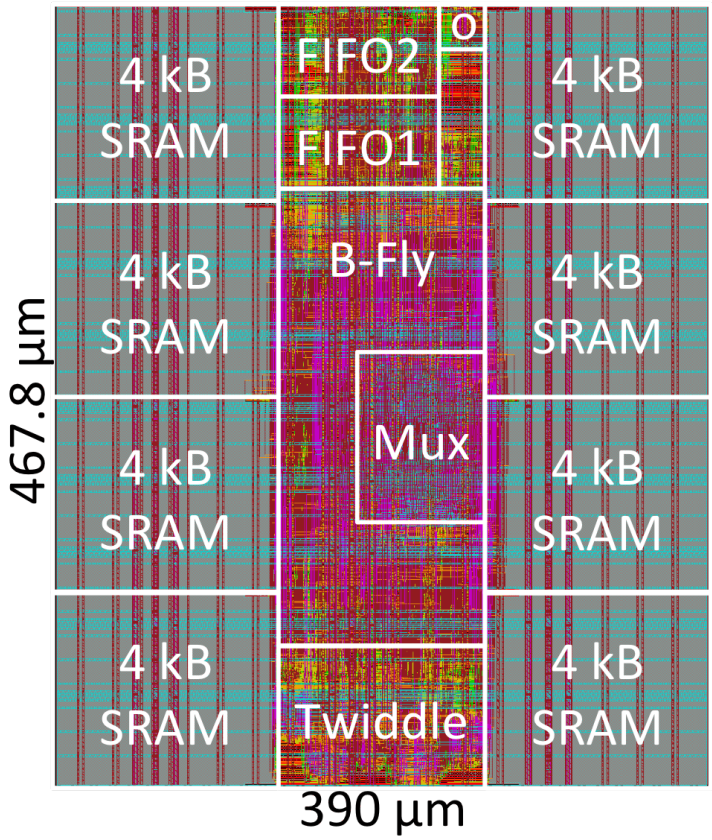


Figure 5.6: Annotated layout of a single KiloCore2 FFT accelerator tile [39].

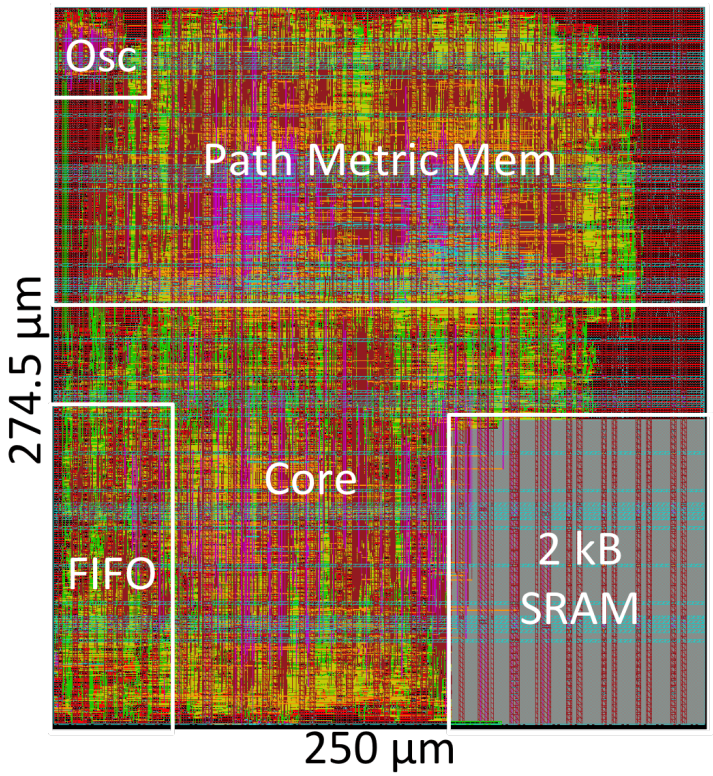


Figure 5.7: Annotated layout of a single KiloCore2 Viterbi accelerator tile [39].

Chapter 6

Software Tools for Writing Many-Core Applications

This chapter discusses the primary tools that were developed to support the KiloCore research effort.

6.1 Many-Core Simulators

6.1.1 AsAP2 Simulator

When this work began, the VCL group at UC Davis was developing AsAP2 applications directly through Verilog simulation. Application code was hand written in assembly, inter-core links were manually set up using network control fields for each tile, and testing was done by visually reading simulation waveforms of processor pipeline activities. Simulations would take hours for small applications like 802.11a (23 processors), up to a month for a full-array record Sorting (164 processors).

To address these inefficiencies, development began on a model of AsAP2's functionality with needless detail abstracted away. Over time, this developed into a full simulator, and was quickly adopted by the VCL group. Several notable features include:

- Written in C++
- Primary goal is to aid in application development, and architected accordingly
- Supports all AsAP2 core types: processor, memory, FFT accelerator, Viterbi accelerator, and Motion Estimation accelerator, as well as external chip input and output handling logic

- Assembly modeled by re-entrant macros which update pipeline state, reentry handled by long jumps
- Assembly functions are optionally parameterized, for easily customizing code on a per-core basis
- Abstracted array configuration, eg. a single function to make a circuit link between two cores regardless of distance
- Supports GALS operation: cores operate on unique clock periods and voltage domains, and may be adjusted during simulation
- Time and energy estimation based on actual AsAP2 measurements, across voltages
- Detailed pipeline model ensures cycle accuracy within a core, and communication between cores is done with sub-cycle accurate data transfers (due to asynchronous clocking)
- Arbitrarily multi-threaded, where each thread handles multiple cores, thread count can be tuned to the host system, and inter-core communication uses custom, thread-safe asynchronous FIFOs (avoids lock, mutex or semaphore overhead)
- Fast run times; a Verilog 1-month Sorting simulation is reduced to less than 1 minute
- Integrates into a host IDE (nominally Visual Studio) for breakpoint support to aid in code debugging
- Automated checks warn on coding hazards (eg. read-after-write), uninitialized memory access, unconnected network ports, and other possible errors
- A wide variety of code profiling functions
- Support for integrating custom C++ code into the assembly simulation, commonly used for additional debug checks or conditional breakpoints
- Support for saving and restoring long simulations
- Support for exploring a variety of DVFS algorithms

6.1.2 KiloCore and KiloCore2 Simulator

Following the tapeout of KiloCore, a new version of the simulator was developed to support the KiloCore architecture and streamline existing functionality. Time and energy estimation were updated with the new chip measurements. Notable new features include:

- New global timing model simplifies modeling of environmental conditions, such as voltage droop or temperature changes
- Naturally models process variation, with some cores being faster than others
- Integrated support for checking data at any link or final output against golden references
- Supports the new packet network, with zero simulation overhead for routers which can never be trafficked
- Improved extensibility for adding custom simulation modules, such as off-chip memories, admin processes, and environmental condition models
- API functionality for integrating with external programs, to control characteristics of the simulation and to return results

Figure 6.1 shows the simulator at work, paused at a breakpoint in the FFT application when computing a butterfly. Here, the contents of the host processor's data memory are displayed, as well as the pipeline contents. All simulated processor state may be viewed in this way by the user.

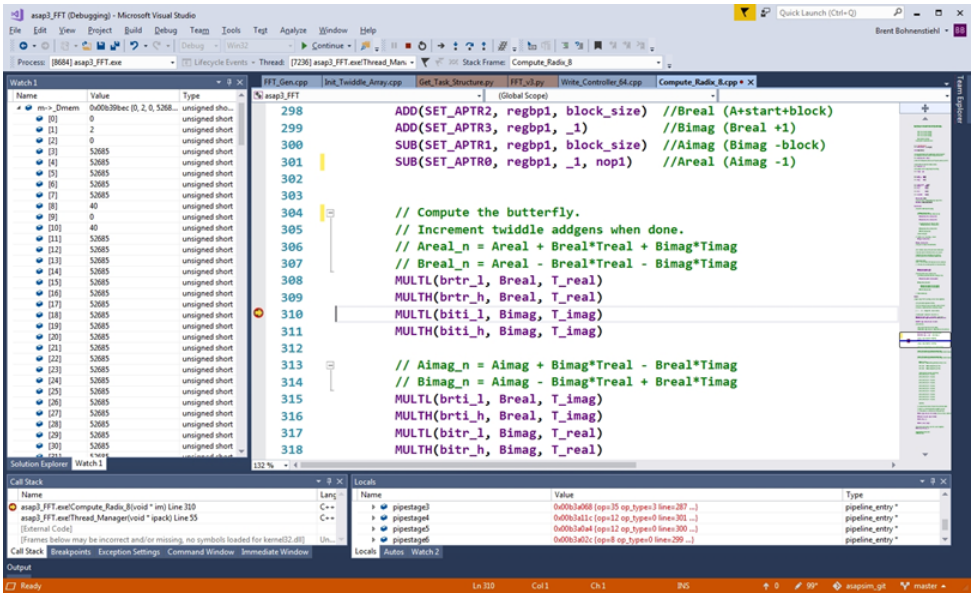


Figure 6.1: KiloCore simulator paused at a breakpoint in the assembly and viewing memory contents, enabled by running in the Visual Studio IDE

These many-core simulators have been a key part of a variety of many-core related research efforts [9, 12–14, 28, 40–48].

6.2 KiloCore Compiler

While the many-core simulators greatly speed up runtime and ease assembly debugging, they still require the programmer to learn the appropriate assembly and architecture characteristics. To make programming more accessible, the latter years of this research include the development of an optimizing compiler to handle assembly generation.

The KiloCore compiler (KCC) is designed as a custom back end to the LLVM compiler infrastructure. Any suitable front end may be used to process programmer input code into the LLVM intermediate representation (IR) format. The back end parses this IR code, and progressively transforms it into a form suitable to KiloCore assembly. Though the LLVM output is targeted at stack-based load/store architectures with separate register files and memory, KCC handles converting such code into a form suitable for a stackless, 16-bit, direct-memory-access architecture. A minimal effort mapping of IR to assembly results in poor code quality, so the bulk of KCC is devoted to analyzing and optimizing the code to KiloCore's particular strengths.

The standard programming flow is to write kernels using C++, with a suitable header library to represent select parts of the KiloCore hardware. At minimum, the programmer must declare and use explicit network ports for data input and output from a processor. Optional parts of the header library support parts of the hardware that are not easily exploited by compiler optimization, such as the address generation units or the 40-bit multiply-accumulation register.

Clang, a well developed front end for LLVM, is used to process the C++ into IR. While off-the-internet distributions of Clang are supported, a lightly customized version of Clang and LLVM is preferred in which Kilocore is specified as using 16-bit pointers and integers.

As IR code can potentially be generated by a variety of front ends, KCC also has limited, proof-of-concept support for handling kernels written in Python, with the help of the Numba package. A runtime patch is applied to Numba to add support for volatile loads and stores, which are utilized to safely model network access operations.

Internally, KCC is organized into a variety of analysis and transform modules. Each module defines its requirements (other modules that must run first) and invalidations-on-change (other modules that need to be rerun if a code change is made). With this, the compiler flow is made self-organizing. This and various other characteristics of KCC were selected to benefit manageability and extensibility for a sole developer.

Figure 6.2 shows a simplified example of code compilation. User supplied input code (a) defines a task which reads input, does some compute, and sends output. This is fed into customized Clang to generate LLVM IR format (b), including any target-nonspecific optimizations LLVM provides. KCC processes this IR into an un-optimized form suitable for assembly generation (c), and optionally performs a wide variety of optimizations to improve code quality (d). Later tools will customize packet headers and IO port numbers based on final task to array mapping and routing.

At the time of this writing, KCC generated code performs to within 7% of carefully hand-crafted and optimized assembly, across all applications that have been converted from assembly to C++. The largest difference occurs in LDPC, which utilizes architectural shortcuts in a critical path loop which do not yet have corresponding compiler optimizations.

6.3 Project Manager

While the simulator and compiler aid in writing code kernels and validating them, they do not particularly help a programmer define and connect the potentially thousands of tasks that will make up a complex application. These tasks also need to be mapped to a target many-core array, routed through the circuit network, and profiled and optimized for ideal frequency and voltage settings.

The KiloCore Project Manager (PM) was written to address these and other needs. Written in Python, it executes a user supplied script which will define the tasks, specify operations to perform (eg. map, simulate, profile), and validate results. The PM acts as a user-facing front end for other KiloCore tools, including the simulator, compiler, and mapper. Some notable features of the PM include:

- Convenient task paramaterization, grouping, and replication (of individuals or groups)
- Automatic data stream splitting and joining for parallel tasks clusters
- An application's tasks may easily be imported into other projects, e.g. the AES engine can be added as a component to any larger application
- Performs task-to-core mapping and routing, either with a basic good-enough-for-simulation algorithm, or through a sophisticated mapping tool written by a co-worker in Julia
- Runs KCC on provided C++ source files, and caches generated assembly for reuse

```
void Add_Inc(core_input  input, core_output output){
  for (short i = 0; i < 8; i++) {
    output << input.read() + i; } }
```

(a) Input C++

```
define dso_local void @_Z7Add_Inc10core_input11core_output(%class.core_input* %input, %class.
    core_output* %output) local_unnamed_addr #4 {
entry:
  %link_ushort.i = getelementptr %class.core_input, %class.core_input* %input, i16 0, i32 0,
      i32 0, i32 6
  %link_ushort.i.i = getelementptr %class.core_output, %class.core_output* %output, i16 0,
      i32 0, i32 0, i32 6
  br label %for.body
for.cond.cleanup:
  ret void
for.body:
  %i.09 = phi i16 [ 0, %entry ], [ %inc, %for.body ]
  %0 = load volatile i16, i16* %link_ushort.i, align 2, !tbaa !27
  %add = add i16 %0, %i.09
  store volatile i16 %add, i16* %link_ushort.i.i, align 2, !tbaa !27
  %inc = add nuw nsw i16 %i.09, 1
  %exitcond = icmp eq i16 %inc, 8
  br i1 %exitcond, label %for.cond.cleanup, label %for.body  }
```

(b) LLVM IR

```
Function( Add_Inc ){
entry:
  MOVE( dmem[128], 0, SW )
  BRL( for_body )
for_cond_cleanup:
  BRL( entry )
for_body:
  MOVE( dmem[0], input_0, nop3, SW)
  ADDU( dmem[0], dmem[0], dmem[128], nop3,SW)
  MOVE( output_0, dmem[0] )
  ADDU( dmem[0], dmem[128], 1, nop3, SW)
  MOVE( dmem[128], dmem[0], SW)
  XOR( null0, dmem[0], 8 )
  BRNZL( for_body )
  BRL( for_cond_cleanup ) }
```

(c) Unoptimized Assembly

```
Function( Add_Inc ){
entry:
RPT(1024)
  ADDU( output_0, input_0, 0 )
  ADDU( output_0, input_0, 1 )
  ADDU( output_0, input_0, 2 )
  ADDU( output_0, input_0, 3 )
  ADDU( output_0, input_0, 4 )
  ADDU( output_0, input_0, 5 )
  ADDU( output_0, input_0, 6 )
  ADDU( output_0, input_0, 7 )
  END_RPT
BRL( entry ) }
```

(d) Optimized Assembly

Figure 6.2: Example of the KiloCore compiler work flow; (a) user provided C++ code (simplified for brevity), (b) LLVM IR format output from Clang, (c) non-optimized translation from IR to KiloCore Assembly, and (d) optimized KiloCore Assembly

- Prepares, builds, and runs the simulator
- Prepares and runs applications on the physical chip (if available)
- Capable of fine-tuning core frequencies through iterative simulations, minimizing energy lost to processor stalls

- Using technology models combined with application profiling, can optimize voltage rail selection for the 3 rails in KiloCore 2 to minimize energy at no loss of throughput
- GUI provides an accessible way for users to run scripts, view task layouts and mappings, run the integrated tools, and view simulation results

Figure 6.3 shows the Project Manager GUI after a simulation of a 650-processor version of the FFT application on KiloCore 2. Simulation metrics are recorded for each individual processor, along with a global summary. Figure 6.4 shows the particular task mapping used in this simulation, with labels hidden due to clutter.

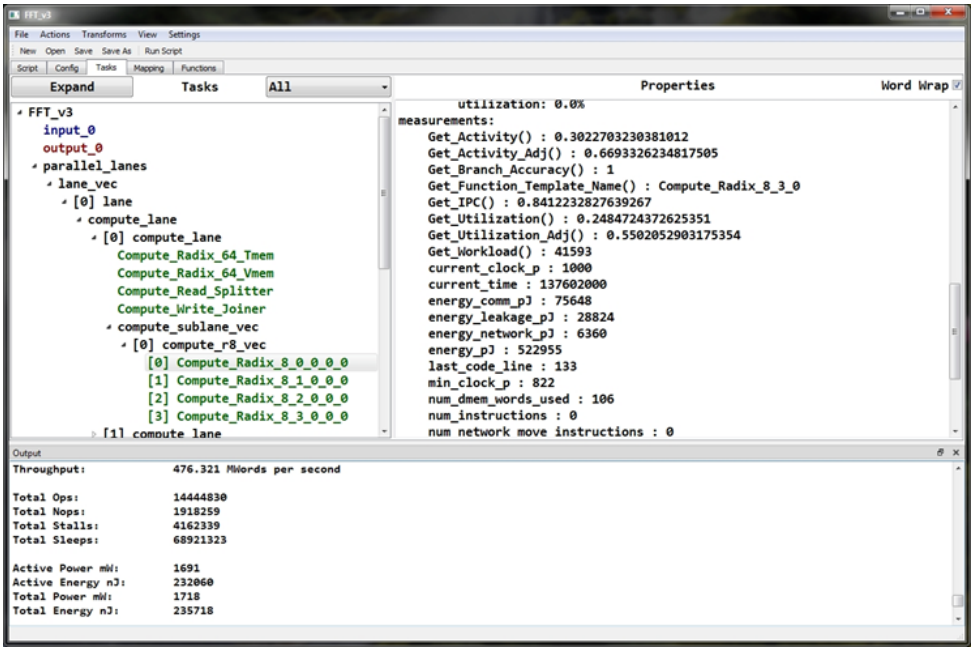


Figure 6.3: KiloCore Project Manager GUI showing a version of the FFT application opened to the tasks tab, with simulation results visible

Table 6.1 summarizes some characteristics of these tools. Any generated code files or external libraries are not included. The KiloCore simulator is somewhat smaller than the AsAP2 simulator due to streamlining and lack of a motion estimation accelerator. Code related to tool chain testing and verification has been omitted. The task mapper is written by a co-worker. The high ratio of source lines to code lines in KCC is due to a high amount of internal documentation important to developing and maintaining an optimizing compiler. The high ratio in the Project Manager is due to API documentation exported to HTML and the GUI for user reference.

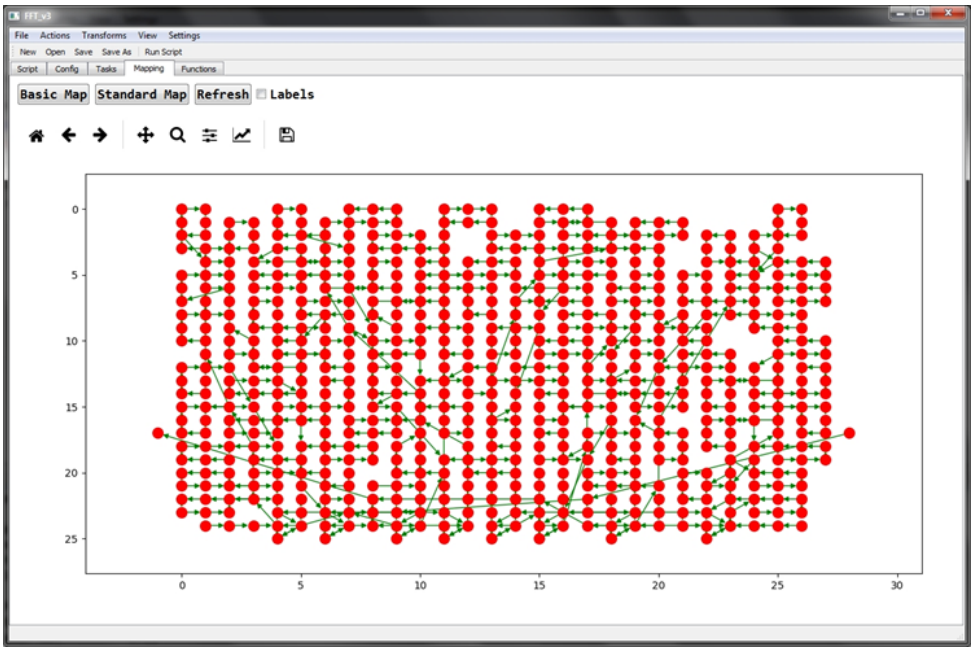


Figure 6.4: KiloCore Project Manager GUI showing a version of the FFT application opened to the mapping tab, which also may be zoomed in and display task names

Table 6.1: Characteristics of several tools developed as part of this research. The compiler metrics include KiloCore C++ header libraries. Metrics do not include testing code or code that isn't original to the KiloCore project.

Tool	Primary Language	Source Lines	Code Lines
AsAP2 Simulator	C++	21670	11730
KiloCore Simulator	C++	19940	10247
Compiler	Python	59797	16869
Project Manager	Python	33572	9506
Task Mapper	Julia	13227	8943

Chapter 7

Conclusion

Key to understanding the operation of many-core arrays is knowledge of how their applications are written and behave. This dissertation opened with the design of such an application: a Low Density Parity Check decoder running on the AsAP2 164-processor architecture. This decoding algorithm is formed from a collection of individual data processing and storage tasks, which were balanced and replicated to fill the processor array.

A detailed exploration was made of the AsAP2 architecture and its applications. Various opportunities for improvement were found, including in the instruction set, pipeline design, network communication between processors, and voltage control logic. To name a few findings: inclusion of unsigned support speeds some operations up by as much as 15x, low-overhead branch prediction raises the correct prediction rate from 27% to 96%, fast oscillator halting reduces active-clock stall cycles by 33%, and voltage dithering improves DVFS energy savings by 16%.

The 1,000 processor KiloCore chip was presented. Fabricated in 32 nm and occupying 64 mm^2, this newly designed architecture implements lessons learned from AsAP2 along with other innovations. KiloCore processors may operate up to 1.78 GHz at 1.1 V, down to 115 MHz at 560 mV where an operation dissipates only 5.8 pJ. In situations with many processors turning on at once, up to 999 of them, the KiloCore oscillators are shown to naturally slow down with voltage grid droop, allowing running processors to remain under their maximum frequency and maintain error-free operation. Four applications are written or adapted for KiloCore, their characteristics and performance discussed. Across these applications and when scaled to the same fabrication technology, KiloCore at 0.9 V achieves geometric mean improvements of 3.1x higher throughput per area and 16.7x higher energy efficiency compared to CPUs and GPUs, with an overall 70.5x improvement in throughput per area per Watt.

Following this, the 697 processor KiloCore2 chip was presented. Fabricated in 32 nm and occupying 64 mm^2, this second generation refinement of the KiloCore architecture is designed to achieve a 63% higher throughput per processor than the original KiloCore, supports three voltage domains that processors may be connected to to optimize energy usage, utilizes a custom chip package to greatly increase the number of I/O ports, and implements a new low-area packet routing network that is specifically designed for the needs of a many-core system. Increased core speeds can reduce compute latency by over 39% for latency sensitive applications. Preliminary estimates are that the array will reach 2.0 Tera-operations per second at 1.1 V.

Finally, the programming and analysis tools for many-core arrays were presented. A high speed simulator, written in C++, is over 50,000 times faster than prior Verilog simulations, and contains a suite of features to aid in application development and debugging. This simulator enabled rapid architectural exploration, leading to the insights that eventually developed into the KiloCore architecture. The KiloCore compiler was presented for generating optimized assembly from user supplied kernels written in C++, Python, or potentially other languages. Leveraging the LLVM infrastructure to act as a front end, this compiler focuses on the back end operations needed to lower LLVM IR code into the format needed for stackless, 16-bit, direct-memory-access processors with strict memory limitations. Supplementing these tools is a Project Manager, which allows users to write simple Python scripts to define a collection of tasks, replicate and map them to a target many-core array, and compile and run their applications.

These hardware and software design techniques are scalable well beyond the 1000 processors in KiloCore. Modern fabrication technology and commercial chip areas have the potential to fit tens of thousands of such processors on a single die. Following in the footsteps of Kilocore, we look forward to more of these thousand-processor-era architectures being developed in the near future.

References

[1] S. Borkar. Thousand core chips: A technology perspective. In *2007 44th ACM/IEEE Design Automation Conference*, pages 746–749, June 2007.

[2] S. Vangal, J. Howard, G. Ruhl, S. Dighe, H. Wilson, J. Tschanz, D. Finan, P. Iyer, A. Singh, T. Jacob, S. Jain, S. Venkataraman, Y. Hoskote, and N. Borkar. An 80-tile 1.28 TFLOPS network-on-chip in 65nm CMOS. In *2007 IEEE International Solid-State Circuits Conference. Digest of Technical Papers*, pages 98–589, Feb 2007.

[3] M. Butts. Synchronization through communication in a massively parallel processor array. *IEEE Micro*, 27(5):32–40, Sep. 2007.

[4] D.N. Truong, W.H. Cheng, T. Mohsenin, Zhiyi Yu, A.T. Jacobson, G. Landge, M.J. Meeuwsen, C. Watnik, A.T. Tran, Zhibin Xiao, E.W. Work, J.W. Webb, P.V. Mejia, and B.M. Baas. A 167-processor computational platform in 65 nm cmos. *Solid-State Circuits, IEEE Journal of*, 44(4):1130–1144, April 2009.

[5] Z. Yu, M. J. Meeuwsen, R. W. Apperson, O. Sattari, M. Lai, J. W. Webb, E. W. Work, D. Truong, T. Mohsenin, and B. M. Baas. AsAP: An asynchronous array of simple processors. *IEEE Journal of Solid-State Circuits*, 43(3):695–705, March 2008.

[6] S. Bell, B. Edwards, J. Amann, R. Conlin, K. Joyce, V. Leung, J. MacKay, M. Reif, Liewei Bao, J. Brown, M. Mattina, Chyi-Chang Miao, C. Ramey, D. Wentzlaff, W. Anderson, E. Berger, N. Fairbanks, D. Khan, F. Montenegro, J. Stickney, and J. Zook. TILE64 - processor: A 64-core SoC with mesh interconnect. In *Solid-State Circuits Conference, 2008. ISSCC 2008. Digest of Technical Papers. IEEE International*, pages 88–598, Feb 2008.

[7] K. Kim. Silicon technologies and solutions for the data-driven world. In *2015 IEEE International Solid-State Circuits Conference - (ISSCC) Digest of Technical Papers*, pages 1–7, Feb 2015.

[8] W. M. Holt. Moore's law: A path going forward. In *2016 IEEE International Solid-State Circuits Conference (ISSCC)*, pages 8–13, Jan 2016.

[9] B. Bohnenstiehl, A. Stillmaker, J. Pimentel, T. Andreas, B. Liu, A. Tran, E. Adeagbo, and B. Baas. KiloCore: A fine-grained 1,000-processor array for task parallel applications. *IEEE Micro*, 37(2):63–69, March 2017.

[10] Andrew Duller, Gajinder Panesar, and Daniel Towner. Parallel processing ? the picoChip way! 01 2003.

[11] M. Butts and A. M. Jones. TeraOPS hardware and software: A new massively-parallel, MIMD computing fabric IC. In *HotChips Symposium on High-Performance Chips*, Stanford, CA, August 2006.

[12] B. Bohnenstiehl, A. Stillmaker, J. Pimentel, T. Andreas, B. Liu, A. Tran, E. Adeagbo, and B. Baas. A 5.8 pJ/Op 115 billion Ops/sec, to 1.78 trillion Ops/sec 32 nm 1000-processor array. In *Symposium on VLSI Circuits*, June 2016.

[13] B. Bohnenstiehl, A. Stillmaker, J. Pimentel, T. Andreas, B. Liu, A. Tran, E. Adeagbo, and B. Baas. KiloCore: A 32 nm 1000-processor array. In *IEEE HotChips Symposium on High-Performance Chips*, August 2016.

[14] B. Bohnenstiehl, A. Stillmaker, J. Pimentel, T. Andreas, B. Liu, A. Tran, E. Adeagbo, and B. Baas. KiloCore: A 32-nm 1000-processor computational array. *IEEE Journal of Solid-State Circuits (JSSC)*, 52(4):891–902, April 2017.

[15] Aaron Stillmaker, Brent Bohnenstiehl, and Bevan Baas. The design of the KiloCore chip. In *ACM/IEEE Design Automation Conference*, Austin, TX, Jun. 2017.

[16] J. Howard, S. Dighe, Y. Hoskote, S. Vangal, D. Finan, G. Ruhl, D. Jenkins, H. Wilson, N. Borkar, G. Schrom, F. Pailet, S. Jain, T. Jacob, S. Yada, S. Marella, P. Salihundam, V. Erraguntla, M. Konow, M. Riepen, G. Droege, J. Lindemann, M. Gries, T. Apel, K. Henriss, T. Lund-Larsen, S. Steibl, S. Borkar, V. De, R. V. D. Wijngaart, and T. Mattson. A 48-core IA-32 message-passing processor with DVFS in 45nm CMOS. In *2010 IEEE International Solid-State Circuits Conference - (ISSCC)*, pages 108–109, Feb 2010.

[17] G. Chrysos. Intel Xeon Phi coprocessor (codename Knights Corner). In *2012 IEEE Hot Chips 24 Symposium (HCS)*, pages 1–31, Aug 2012.

[18] B. D. de Dinechin, R. Ayrignac, P. Beaucamps, P. Couvert, B. Ganne, P. G. de Massas, F. Jacquet, S. Jones, N. M. Chaisemartin, F. Riss, and T. Strudel. A clustered manycore processor architecture for embedded and accelerated applications. In *2013 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–6, Sep. 2013.

[19] A.T. Jacobson, D.N. Truong, and B.M. Baas. The design of a reconfigurable continuous-flow mixed-radix FFT processor. In *Circuits and Systems, 2009. ISCAS 2009. IEEE International Symposium on*, pages 1133–1136, May. 2009.

[20] Bin Liu and B.M. Baas. Parallel AES encryption engines for many-core processor arrays. *Computers, IEEE Transactions on*, 62(3):536–547, March 2013.

[21] Aaron Stillmaker, Lucas Stillmaker, and Bevan Baas. Fine-grained energy-efficient sorting on a many-core processor array. In *Parallel and Distributed Systems (ICPADS), 2012 IEEE 18th International Conference on*, pages 652–659, Singapore, Singapore, December 2012.

[22] A.T. Tran, D.N. Truong, and B.M. Baas. A complete real-time 802.11a baseband receiver implemented on an array of programmable processors. In *Signals, Systems and Computers, 2008 42nd Asilomar Conference on*, pages 165–170, Oct 2008.

[23] Zhibin Xiao, Stephen Le, and Bevan Baas. A fine-grained parallel implementation of a H.264/AVC encoder on a 167-processor computational platform. In *IEEE Asilomar Conference on Signals, Systems and Computers (ACSSC)*, November 2011.

[24] R.G. Gallager. Low-density parity-check codes. *Information Theory, IRE Transactions on*, 8(1):21–28, January 1962.

[25] Jinghu Chen, A. Dholakia, E. Eleftheriou, M.P.C. Fossorier, and Xiao-Yu Hu. Reduced-complexity decoding of LDPC codes. *Communications, IEEE Transactions on*, 53(8):1288–1299, Aug 2005.

[26] Rongchun Li, Jie Zhou, Yong Dou, Song Guo, Dan Zou, and Shi Wang. A multi-standard efficient column-layered LDPC decoder for software defined radio on GPUs. In *SPAWC, 2013*, pages 724–728, June 2013.

[27] Aaron Stillmaker, Zhibin Xiao, and Bevan Baas. Toward more accurate scaling estimates of CMOS circuits from 180 nm to 22 nm. Technical Report ECE-VCL-2011-4, VLSI Computation Lab, ECE Department, University of California, Davis, December 2011. http://www.ece.ucdavis.edu/cerl/techreports/2011-4/.

[28] Jon J. Pimentel and Bevan M. Baas. Hybrid floating-point modules with low area overhead on a fine-grained processing core. In *IEEE Asilomar Conference on Signals, Systems and Computers (ACSSC)*, Pacific Grove, CA, November 2014.

[29] Zhiyi Yu and Bevan M. Baas. A low-area multi-link interconnect architecture for GALS chip multiprocessors. *Very Large Scale Integration (VLSI) Systems, IEEE Trans. on*, 18(5):750–762, May 2010.

[30] A. T. Tran and B. M. Baas. Achieving high-performance on-chip networks with shared-buffer routers. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 22(6):1391–1403, June 2014.

[31] R. W. Apperson, Z. Yu, M. J. Meeuwsen, T. Mohsenin, and B. M. Baas. A scalable dual-clock FIFO for data transfers between arbitrary and haltable clock domains. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 15(10):1125–1134, Oct 2007.

[32] Zhiyi Yu and Bevan M. Baas. High performance, energy efficiency, and scalability with GALS chip multiprocessors. *IEEE Trans. on Very Large Scale Integration Systems*, 17(1):66–79, January 2009.

[33] D. M. Chapiro. *Globally-asynchronous locally-synchronous systems*. PhD thesis, Stanford University, Stanford, CA, USA, 1984.

[34] K. Iwai, T. Kurokawa, and N. Nisikawa. AES encryption implementation on CUDA GPU and its analysis. In *Networking and Computing (ICNC), 2010 First International Conference on*, pages 209–214, Nov 2010.

[35] Q. Li, C. Zhong, K. Zhao, X. Mei, and X. Chu. Implementation and analysis of AES encryption on GPU. In *High Performance Computing and Communication 2012 IEEE 9th International Conference on Embedded Software and Systems (HPCC-ICESS), 2012 IEEE 14th International Conference on*, pages 843–848, June 2012.

[36] Xia Pan, Xiao fan Lu, Ming qi Li, and Rong fang Song. A high throughput LDPC decoder in CMMB based on virtual radio. In *Wireless Communications and Networking Conference Workshops (WCNCW), 2013 IEEE*, pages 95–99, April 2013.

[37] G. Wang, M. Wu, B. Yin, and J. R. Cavallaro. High throughput low latency LDPC decoding on GPU for SDR systems. In *Global Conference on Signal and Information Processing (GlobalSIP), 2013 IEEE*, pages 1258–1261, Dec 2013.

[38] Mike Butler. AMD Bulldozer Core - a new approach to multithreaded compute performance for maximum efficiency and throughput. In *IEEE HotChips Symposium on High-Performance Chips (HotChips 2010)*, August 2010.

[39] Aaron Stillmaker. *Design of Energy-Efficient Many-Core MIMD GALS Processor Arrays in the 1000-Processor Era*. PhD thesis, University of California, Davis, Davis, CA, USA, December 2015. http://vcl.ece.ucdavis.edu/pubs/theses/2015-1.stillmaker/.

[40] Aaron Stillmaker, Lucas Stillmaker, Brent Bohnenstiehl, and Bevan Baas. Energy-efficient sorting on a many-core platform. In *Technology and Talent for the 21st Century (TECHCON 2013)*, September 2013.

[41] Bin Liu, Brent Bohnenstiehl, and Bevan M. Baas. Scalable hardware-based power management for many-core systems. In *IEEE Asilomar Conference on Signals, Systems and Computers (ACSSC)*, Nov. 2014.

[42] Soheil Ghiasi Bin Liu, Mohammad H. Foroozannejad and Bevan M. Baas. Optimizing power of many-core systems by exploiting dynamic voltage, frequency and core scaling. In *IEEE International Midwest Symposium on Circuits and Systems (MWSCAS)*, Aug. 2015.

[43] Emmanuel O. Adeagbo and Bevan M. Baas. In *Technology and Talent for the 21st Century (TECHCON 2015)), title = Energy-Efficient String Search Architectures on a Fine-Grained Many-Core Platform, year = 2015, month = sep.*

[44] B. Bohnenstiehl and B. Baas. A software LDPC decoder implemented on a many-core array of programmable processors. In *2015 49th Asilomar Conference on Signals, Systems and Computers*, pages 192–196, Nov 2015.

[45] Jon J. Pimentel, Brent Bohnenstiehl, and Bevan M. Baas. Hybrid hardware/software floating-point implementations for optimized area and throughput tradeoffs. *IEEE Transactions on Very Large Scale Integration Systems (TVLSI)*, 25(1):100–113, January 2017. Official date of publication July 12, 2016.

[46] Emmanuel O Adeagbo. Energy-efficient pattern matching methods on a fine-grained many-core platform. Master's thesis, University of California, Davis, Davis, CA, USA, March 2017. http://vcl.ece.ucdavis.edu/pubs/theses/2017-1.Adeagbo/.

[47] Peiyao Shi. Sparse matrix multiplication on a many-core platform. Master's thesis, University of California, Davis, Davis, CA, USA, December 2018. http://vcl.ece.ucdavis.edu/pubs/theses/2018-1.pshi/.

[48] Filipe Borges. AlexNet deep neural network on a many core platform. Master's thesis, University of California, Davis, Davis, CA, USA, 2019.