Design of Energy-Efficient Many-Core MIMD GALS Processor Arrays in the 1000-Processor Era

By

AARON THOMAS STILLMAKER
B.S. (California State University, Fresno) 2008
M.S. (University of California, Davis) 2013

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

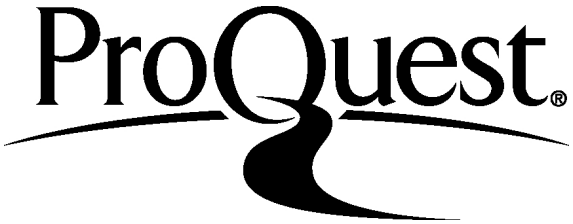
Approved:

Dr. Bevan M. Baas, Chair

Dr. Venkatesh Akella

Dr. Rajeevan Amirtharajah

Committee in charge
2015



Abstract

As transistor sizes continue to scale, more transistors are able to be used in a fixed die size. The recent trend for general purpose processing units is to use the increased number of transistors from process technology scaling to add more processing cores on a single die. At a certain point, it becomes untenable to continue to add more cores with traditional architectures and communication systems, which necessitates a fundamental change in architectures to facilitate these cores. This paradigm shift requires new energy-efficient, high-performance algorithms and hardware designs tailored for many-core processor arrays, as they provide different challenges than a single or multi-core chip. With such large arrays of processors, communication, both between processors and to memories, becomes a limiting factor, requiring algorithms to work with these limitations as well as on-chip interconnect networks to make communication possible.

This dissertation offers three different novel methods to perform a high throughput energy-efficient database sort data records using a fine-grained many-core processor array. When measured against sorts created to fairly compare results, the most energy efficient first-phase many-core sort requires over 83× lower energy than a quick sort performed on an Intel laptop-class processor and over 105× lower energy than a radix sort running on an Nvidia GPU. In addition, the highest first-phase throughput many-core sort is over 10× faster than the quick sort and over 14× faster than the radix sort. Both phases of an entire 10 GB external sort require 6.9× lower energy×time (energy delay product, EDP) than the quick sort and over 13× lower energy×time than the radix sort. The proposed sorts are easily programmed and scalable to any sized 2D mesh processor array while giving a large energy savings without penalizing performance.

The dissertation presents the developed physical design flow and design methodology for creating a digital chip in the 1000-processor era. A number of design considerations are discussed, including module design, power grid design, power gate system design, physical DVFS requirements, communication, and chip level layout.

The design for both KiloCore and KiloCore2 are covered, as well as preliminary measured results from KiloCore, the first fabricated chip containing 1000 MIMD, programmable, independent processing cores on a single die. Early results show that KiloCore can perform at 5.8 pJ/Op at 115 Billion Ops/sec at 0.56 V, and up to 1.78 Trillion Ops/sec at 1.1 V. KiloCore2 contains 697 programable processors, two of which are optimized for high speed, one fast Fourier transform accelerator, and two Viterbi decoder accelerators. Both chips were fabricated in 32 nm partially depleted silicon-on-insulator (PD-SOI) technology. KiloCore2 contains multiple power rails, which allows individual cores to select a voltage based on its workload to save energy, with minimal voltage droop and minimal area.

Acknowledgments

My journey through the PhD process has been difficult, enlightening, and rewarding, and it certainly couldn't have been done without the support of a number of people. I would like to gratefully thank my advisor and mentor, Dr. Bevan Baas, for all of his help and guidance throughout my doctoral research, teaching assignments, and overall education. Beyond imparting me with a broad knowledge of VLSI design and processor architecture, through his patient wisdom, I was fortunate enough to be taught how to execute novel research, how to be an academic, and how to be an instructor. His enthusiasm and endless knowledge has helped set an excellent example for me as I start my career.

I would like to thank Dr. Venkatesh Akella for sharing his architecture knowledge with me during my education, as well as his guidance supervising my teaching assistantships, and participating in my PhD guidance committee, qualification exam committee, and dissertation reading committee. I would like to thank Dr. Rajeevan Amirtharajah for advancing my knowledge of digital circuits, as well as serving on my dissertation reading committee. I would also like to thank Dr. S. J. Ben Yoo, Dr. Hussain Al-Asaad, and Dr. Matthew Farrens for being on my qualification exam committee, and their guidance in the qualification exam process, as well as Dr. Soheil Ghiasi for participating in my PhD guidance committee. I would also like to thank all of the faculty and staff of the Electrical and Computer Engineering Department at UC Davis for all of their support.

I would especially like to thank the co-authors in papers and presentations from the VLSI Computation Laboratory that I have made in my tenure at UC Davis, much of which is presented in this report: Dr. Bevan Baas, Jon Pimentel, Brent Bohnenstiehl, Lucas Stillmaker, Bin Liu, Timothy Andreas, Emmanuel Adeagbo, Michael Braly, Anh Tran, and Zhibin Xiao. I want to thank all of the VCL members for their help and discussion especially (the members not listed above): Jeremy Webb, Tinoosh Mohsenin, Nima Mostafavi, Satyabrata Sarangi, Shifu Wu, and Mark Hildebrand. It has been a pleasure to work with all of these colleagues during my time at UC Davis.

I would like to thank my colleagues during my time at Intel, Ram Krishnamurthy, Gregory Chen, Mark Anders, and Himanshu Kaul for broadening my horizons into the industry. I would also like to thank my friend and computer memory architecture researcher, Ameen Akel, for his insightful discussions and advice.

I would especially like to thank my wife, Kimberly Stillmaker, for all of her help and encouragement in my studies and especially for all of tireless nights editing my papers for grammar. Without her understanding support, our chips never would have been made, and my papers would have been hopeless. I am very grateful for my colleague and brother, Lucas Stillmaker, with whom we started the many-core sorting research. I would also like to thank the rest of my family: Dan Stillmaker, Laurie Stillmaker, and Kelsey Stillmaker, as well as my daughters, Lauren Stillmaker and Margaret Stillmaker, for all of their love and support.

Finally, I gratefully acknowledge support from the UC Davis ECE Department, UC Davis GSA, ST Microelectronics, C2S2 Grant 2047.002.014, NSF Grant 0430090 and CAREER Award 0546907, SRC GRC Grant 1598, 1971, and 2321 and CSR Grant 1659, Intel, UC Micro, Intellasys, SEM, and a UCD Faculty Research Grant.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

At the 50th anniversary of the inception of Moores law [1], the observation that the number of devices per chip doubles roughly every two years will continue, at least through the foreseeable future [2, 3], and currently CMOS fabrication technologies allow the creation of dies with billions of devices on a single chip. At the same time, there are concerns that simply scaling transistor sizes will not be enough to satisfy energy and performance demands in the sub-10 nm regime [2, 4, 5]. Nano-electro-mechanical (NEM) devices [6], carbon nanotube transistors (CNTs) [7], and nanowire transistors [8, 9] appear to be promising post-CMOS technologies, but they are years from commercial integration, and still do not address the issues that arise from communication with a growing number of devices. This, coupled with the demands for big-data, the internet of things (IoT), and mobile healthcare, will require increasing performance and energy efficiency [2, 3, 4, 10].

Traditional processor architecture designs with a fixed die size are untenable with continued transistor scaling due to increasing memory access power [2] and powering a larger number of transistors, which will combine to push such a chip over a reasonable power envelope, due largely to thermal concerns [2, 12]. Yet, the performance and efficiency gains possible through parallel processing have been known for many years [14, 15, 16]. This necessitates a paradigm shift in the fundamental architecture of processors. There have been explorations into the next era of computing, looking at neural-networks [17], and fine-grained many-core processors [18]. This dissertation explores the latter, and heralds the coming of a 1000-processor era of computing, shown

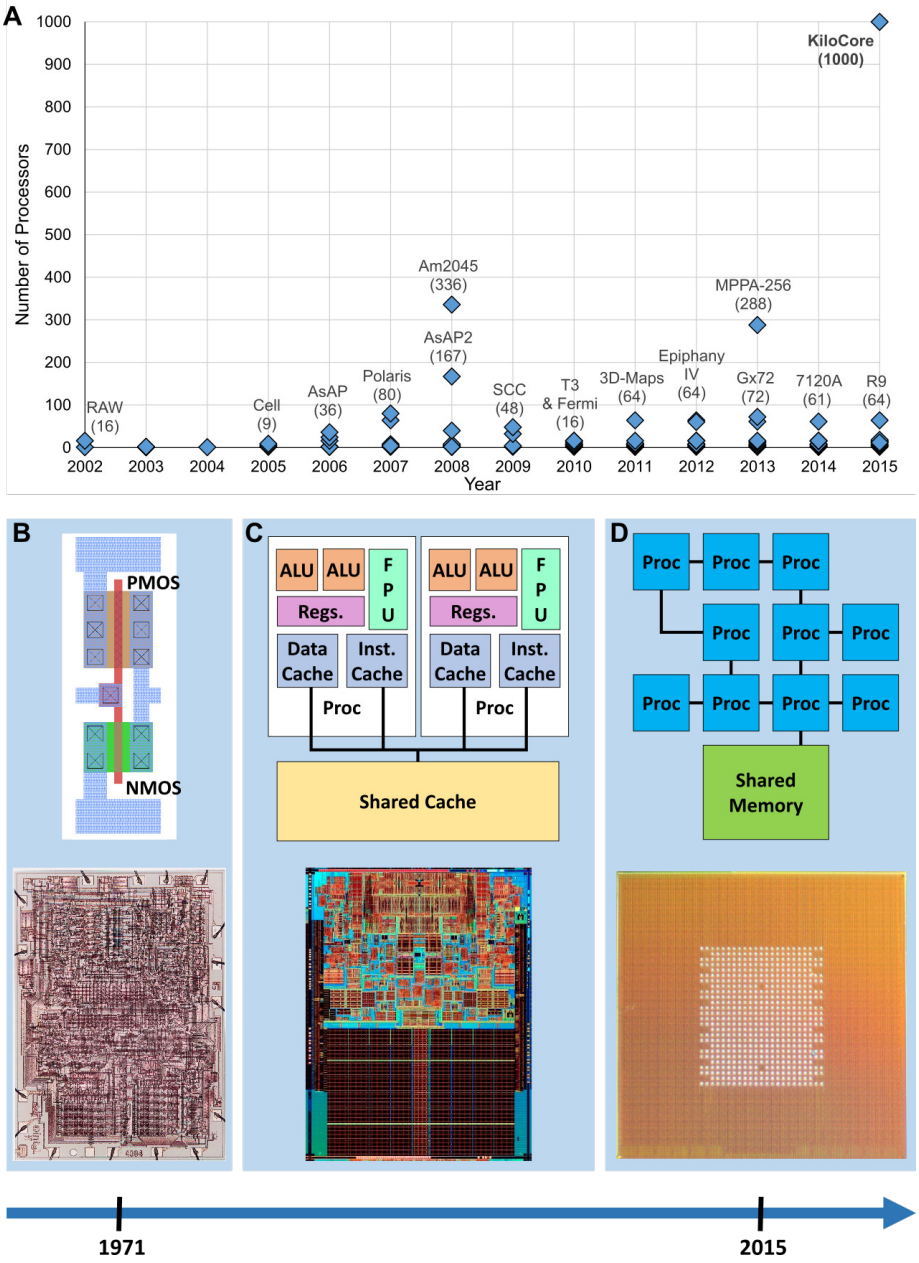


Figure 1.1: Progressions of digital design eras over time. (A) Number of processors capable of independent program execution on a single die versus year of introduction or publication for fabricated chips. (B) Transistor Era: the Intel 4004 was the first commercial single-chip microprocessor and contained 2300 hand-drawn transistors [11]. (C) Single/Multi-Processor Era: design efforts focus on components of single processors and multi-processors, which generally scale well to only small numbers of processors [12, 13]. (D) 1000-Processor Era: design efforts focus on making systems scalable and working with processors as building blocks. Our 32 nm 1000-processor chip would contain approximately 2300-3700 processors if its area were the same as a 32 nm Intel Core i7 processor.

in Figure 1.1, with the first ever fabricated chip with 1000 independent programmable processors on a single die. A number of chips have been marketed as having more than 1000 cores, such as some recent graphical processing units (GPUs) [19], but these single instruction, multiple data (SIMD) arrays cannot operate independently of a controller, and as such, are not defined as separate processing cores. Modern field-programable gate array (FPGA) devices advertise up to 5.5 million logic elements, each consisting of a look-up table (LUT), a register and additional logic, and can be flexibly rewired to create custom circuits, at a relatively high energy and area cost [20]. As such, while FPGAs offer an interesting design point, they are not considered processor arrays. The first commercially available single-chip microprocessor was the Intel 4004 [11], with 2300 transistors on a single chip. In the presented 64mm chip, there are 1000 processors, but it would only take a 150mm chip (smaller than many current Intel Core processors) to contain more than 2300 processors on a single chip.

In this new era of computing, there are a number of large departures from traditional architecture, spurred by the realities of sub-10 nm transistors in the near future. Many-core processors have recently been shown to be a promising solution to rising energy-efficiency and performance demands [12, 18, 21, 22, 23, 24, 25], but they have not been proven a viable solution with hundreds of processors on a single die, until now.

Energy efficiency in large database data centers is an increasingly important concern [26, 27]. In the United States during 2006, data centers used 1.5% of the country's total power consumption, costing a total of $4.5 billion [28]. Processing units are a large contributing factor to a system's overall power usage, both directly and indirectly through required cooling. There is a demand for reduced power consumption while not reducing processing ability for both cost and environmental concerns [29]. Historically server class processors have valued throughput over energy cost. However the general trend in database servers over the last 10 years has been for performance to increase much faster than performance per watt [30].

Sorting is one of the most used processing kernels in database systems [31], creating an interest in energy conscious sorting methods [32]. Datacenters with large data sets generally perform *external sorts*, where data cannot fit in volatile memory necessitating two separate sorting phases [33].

With these larger arrays scaling to hundreds of cores, current sorting algorithms designed

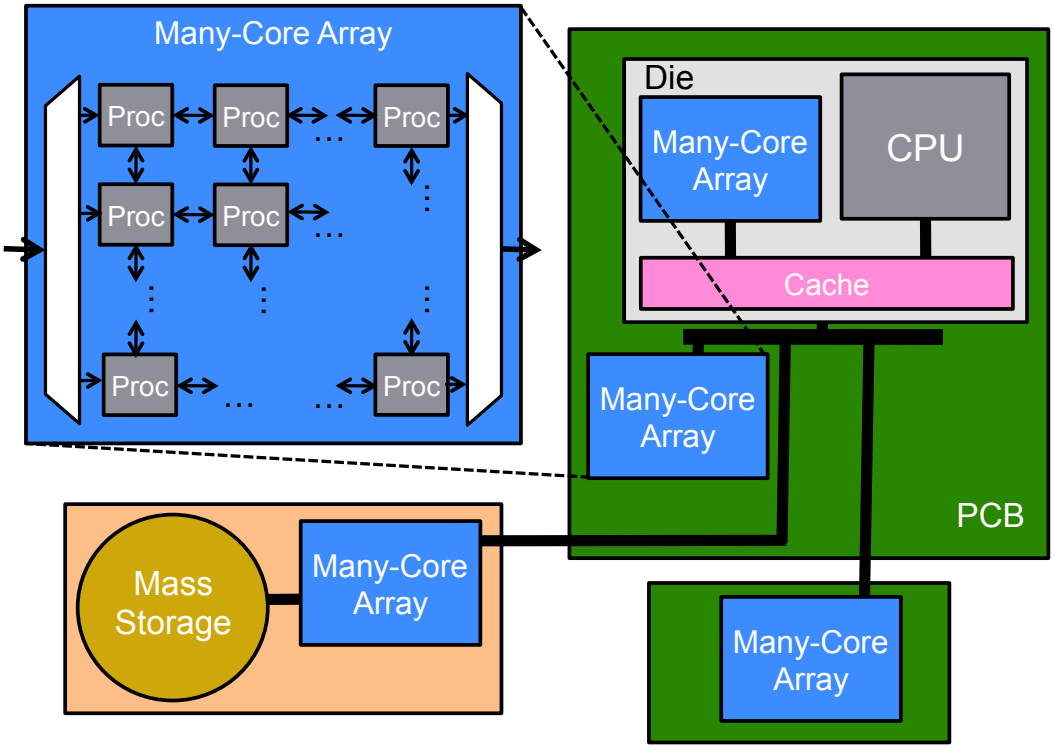


Figure 1.2: Diagram showing a fine-grained many-core architecture with nearest-neighbor communication, chip data input on the left side of the array, chip data output on the right, and no global shared memory with possible implementation locations as a co-processor in a system.

for traditional architectures cannot be used due to differences with architectural features such as intra-processor communication, shared memories, and off chip I/O. As fine-grained many-core processor arrays aren't well suited for all applications, it is proposed that these many-core processor arrays are at first implemented as a co-processor inside a computation system. Figure 1.2 shows a generic view of the fine-grained many-core architecture and different ways that it could be connected into a computer system.

As a case study, this proposal presents energy-efficient, scalable sorting algorithms for an external sort, with the first phase completed by a fine-grained many-core array of low-powered, simple multiple instruction, multiple data (MIMD) processors. The sorts consist of small modular program kernels operating on each core, making them scalable to different array sizes. A many-core array serving as a co-processor performs the first phase of the sort, while an Intel central processing unit (CPU) performs the second phase. The merge sort does not benefit from parallelization onto a large many-core array on a single chip since the computation is I/O bound.

With the progression towards many-core arrays, this report discusses how to design many-core hardware in the 1000-processor era. Specifically we present the design and physical design methodology of two globally asynchronous, locally synchronous (GALS) MIMD processor arrays, KiloCore and KiloCore2. Current computer-aided design (CAD) tool flows for VLSI design are written to support the design of VLSI systems which are currently being designed. This necessitates new tool flows and VLSI design methodology to deal with physically designing large scale chips with hundreds to thousands of repeated complex, independent, homogeneous processing elements. An extra layer of complication arises form communication in a large scale system with GALS operation, of which the implementation has not been widely explored [34]. This dissertation presents the design flow used in the design of KiloCore and KiloCore2, which helps address these two above mentioned concerns. In the presented design methodology, the implementation of power gates as a means to dynamically switch between multiple power rails is discussed.

1.2 Dissertation Organization

The dissertation is organized as follows. Chapter 2 contains background information, covering database sorting, parallel sorting algorithms, transistor performance scaling, and many-core processor design. Chapter 3 presents the kernels that will make up the proposed many-core sorting methods, before going into analysis of the results. First, phase 1 results are analyzed, then the results from a simulated entire 10 GB sort on a many-core array is given. Chapter 4 presents the physical design methodology and flow to design GALS processor arrays in the 1000-processor era, as well as details on a dual-clock first in first out (FIFO) buffer designed to be used in a many-core processor array with multiple clock domains. Chapter 5 covers the design specifics of both KiloCore and KiloCore2, as well as some preliminary measured results from KiloCore. Finally, Chapter 6 gives a summary and conclusion, then describes ongoing and proposed future projects.

Chapter 2

Background

2.1 Sorting

As computers changed to multi-core processors, sorting research adapted, parallelizing sorting algorithms to take advantage of multiple processing cores. The progression to many-core processors [21, 22, 35] necessitates a new shift to sorting with large processor arrays.

2.1.1 Related Work

Internal sorting is the sorting of data sets which can fit completely in main memory. Common examples include the merge, quick, radix, and bitonic sorts [36].

The merge, quick, and radix sorts are computationally simple, and in a processor array, each parallel processor either computes independent parts of a list or is mapped into a data path to perform a part of a sort and then pass the data to another processing stage. Using clever methods of partitioning and redistribution, parallel sorts show promising results [37, 38]. Specific parallel sorting algorithms, such as the odd-even and bitonic merge sorts, were explicitly developed to exploit parallelism [39, 40, 41].

These methods rely on each processor's ability to efficiently access system input/output (I/O), a shared memory or processors other than their neighbors. This is a reasonable expectation for a small array, but not always the case for large many-core arrays.

The SIMD GPUTerraSort takes advantage of large shared memories to attain high throughput [42]. However, with the high power of the GPU and its memory system, the sort is not

energy-efficient.

Theoretical and early work on sorting with mesh-connected processor arrays on arbitrarily-sized arrays is not implementable on our targeted fine-grained many-core architecture because these theoretical discussions target a specific and incompatible architecture to our own and therefore cannot be compared with our results [43, 44]. Sorting on systolic arrays is data driven, similar to our implementation, but the arrays need to either operate in lock step or as a wavefront array with required handshaking [35]. Because of major architectural differences including different communication methods, memory system differences, and timing differences, these ideas cannot be transposed onto a fine-grained many-core array, and with little recent work in the field, no comparison with this work can be made.

Work has been done on sorting on a fine-grained many-core architecture, namely the development of SAISort [45], and the implementation of the SAISort kernel in sorting on a many-core array on the asynchronous array of processors, version 2 (AsAP2) chip to perform the first phase of an external sort [46]. This paper implements these sorts, as well as a new sorting application, different output protocols, scaling on an arbitrarily-sized general many-core platform, and reporting on an entire external sort, instead of just the first phase.

JouleSort [32] is a system-level benchmark for energy efficiency of a large external database sort. The JouleSort benchmark compares total system power of a physical system, while the proposed many-core sorts are designed to work on a simulated co-processor. Therefore the proposed sorts could not utilize the JouleSort benchmark directly, though many details from the guidelines were used and a comparison is made with the processing power of a comparable JouleSort winner in Section 3.5.2.

Vitter [47] utilizes common sorting methods using different I/O paradigms on general purpose CPUs instead of a many-core array to complete an external sort. CloudRAMSort [48] used a large cloud of distributed high performance processors that perform an external sort entirely in DRAM. CloudRAMSort utilizes distributed traditional CPUs as well as SIMD arrays that maximizes performance. Neither of these sorts report data which could be meaningfully compared to the presented sorts.

Work by Solomonik and Kale [36] analyzes the scalability of parallel sorting algorithms on large many-core processor arrays in large distributed systems. They sort 64-bit keys and perform

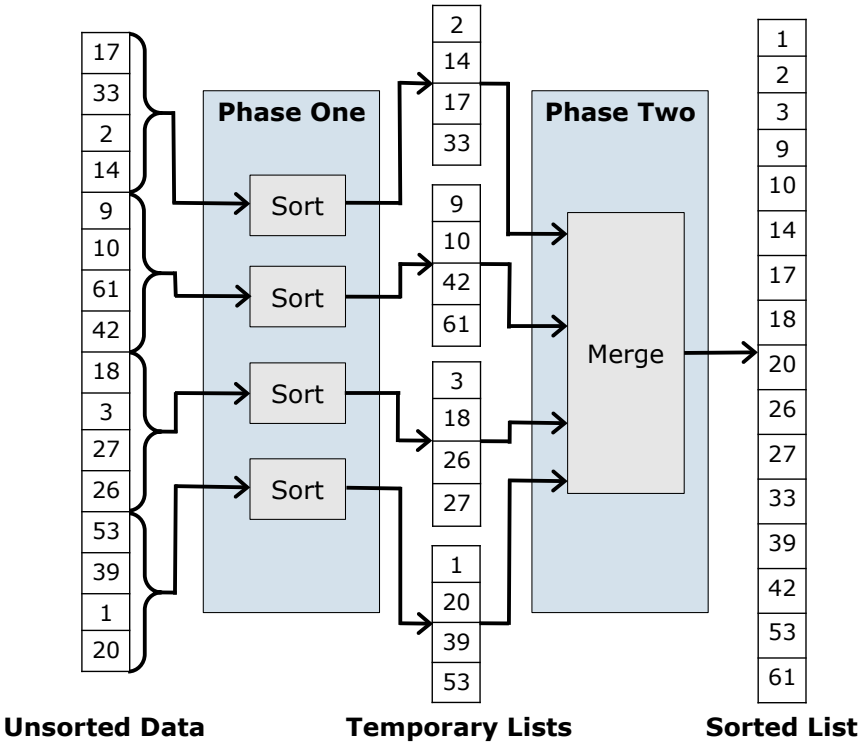


Figure 2.1: Example of an external sort, where 16 unsorted numbers are sorted into 4 separate temporary sorted lists in phase 1, and merged together to form one sorted list. In this example, high-speed local memory is presumed to be of a size such that a maximum of 4 records can be sorted at a time.

an internal sort, which cannot compare with our external sort.

2.1.2 External Sorting

External sorting requires the use of a secondary storage system, like a hard drive, and is done in two phases. During the first phase, temporary sorted lists are created from unsorted data, which can fit inside main memory. In the second phase, these temporary lists are merged together to create one complete sorted list, as shown in Figure 2.1. There are many efficient merging sorts on multi-core processors [37, 49]. We implemented both first and second phases on a many-core array, and it was found the more complex first phase is computationally bound, while the second merging phase is generally I/O bound. Therefore, we focus on the first phase for a many-core implementation, and use a merge sort on a laptop-class processor to model the entire external sort.

2.1.3 Scaling to Many-Core

Challenges of implementing parallel sorts on large processor arrays include balancing the computational load amongst the processors and transmitting the data between processors and memories. Because of these challenges, most sorts don't scale well with more cores [36]. To get around these issues, many parallel processor sorts, such as GPU sorting algorithms, take advantage of shared caches or shared memories to easily access a list, and have many different processors work on different parts of one list.

Most commonly-used multi-core sorts do not scale well to a many-core architecture because they require access to a large shared memory, or high speed networks to allow for shifting and swapping of records, as in the bitonic merge sort. These more complex sorts are not suited for many-core array architectures, and are not explored in this paper. Our target architecture is a low power, small area, many-core system with 10's to 1000's of cores per chip [50].

2.2 Streaming Many-Core Co-Processor Sorting Platform

This paper presents sorting with a large array of processors that have communication only with nearest neighbors and limited long-distance communication. Only processors on the edge of the array have access to chip I/O and the chip contains no explicit global shared memory. The proposed sorts are designed to use a many-core array as a co-processor working in tandem with a general purpose CPU, allowing it to use its complex high power circuits on more appropriate computations.

With only local communication and arbitrarily large arrays, the design of the proposed many-core phase 1 sorts was limited to streaming data through an array. Therefore the co-processor could be used to sort data as it streams from memory to the general purpose CPU or another memory, not involving the general purpose CPU during the first phase.

The results from the first phase of the sort exploration were measured from AsAP2, developed at the University of California, Davis, VLSI Computation Laboratory by Truong et al. [51]. A block diagram showing the AsAP2 chip, highlighting the communication, can be found in Figure 2.2.

For the entire external sort, the proposed sorts are modeled on arrays as they scale upwards of thousands of processors on one array. While this is not currently a viable production option,

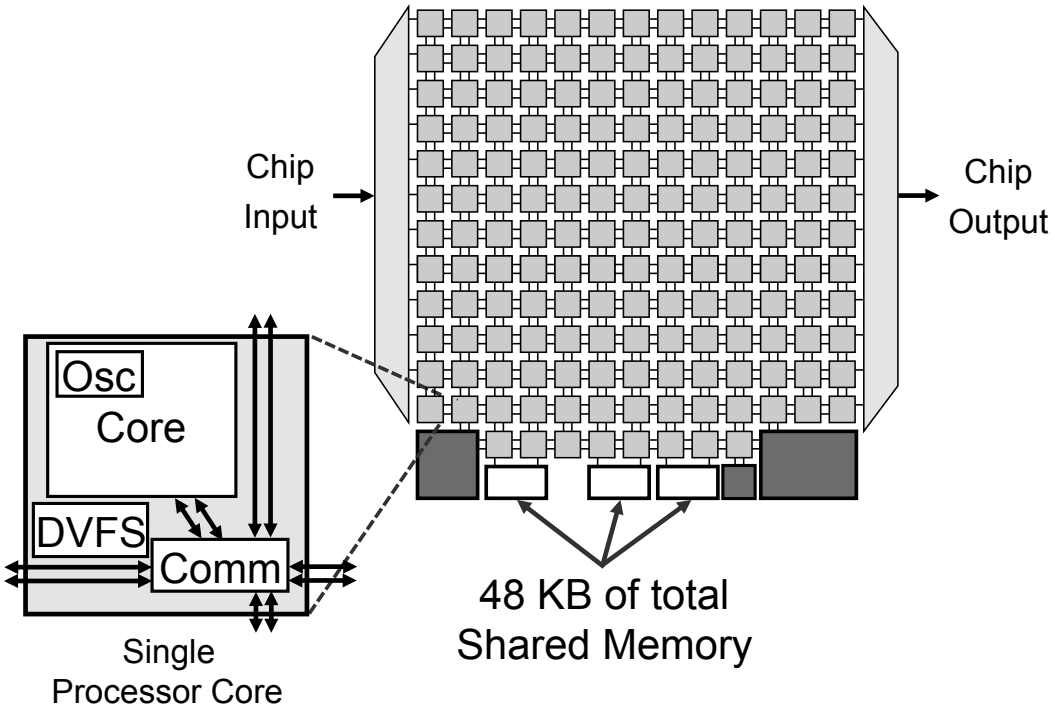


Figure 2.2: Block diagram of the AsAP2 chip, including 164 general purpose processors, 3 special purpose processors, and 3 shared memories. A single processing core is highlighted, showing the communication between processors.

it was interesting to explore how the sorts perform with various array sizes. In order to show quantitative results, it was necessary to use some measured values from an existing architecture. Therefore, the AsAP2 architecture [51] was used for certain limitations and measured values. Each processor contains its own clock domain and oscillator, and has the ability to turn off its oscillator when stalled, which causes active power dissipation to fall to zero (leakage only) while a processor is idle [20]. Processors are able to individually change their supply voltage to further reduce power dissipation [52], but this feature is unused in this work. Processors are connected by a 2D circuit-switched mesh network, allowing for nearest-neighbor communication to adjacent processors [53], as well as long-distance communication that bypasses cores to connect non-adjacent cores [54]. Using the AsAP many-core architecture, we were limited to its reduced instruction set architecture with sixty-three instructions, 128 words of instruction memory, 256 bytes of data memory, and two 128-byte FIFO buffers, for inter-processor communication across clock boundaries per processor. For modeling purposes we used the physically measured traits from the fabricated 65-nm chip, where each processor takes up $0.17\,\mathrm{mm}^2$ of area [18].

2.3 Performance Scaling Factors of Digital CMOS Devices

The observation known as Moore's law [1], which states that the number of devices per chip doubles roughly every two years, continues to hold true [2, 3], largely due to the size of transistors continuing to scale smaller and smaller [55]. Unfortunately, it has become apparent that traditional scaling of performance metrics of CMOS devices does not hold with deep submicron technology sizes [56, 57, 58]. It is, however, still desirable to compare CMOS circuit performance results between circuits that are fabricated using different transistor sizes and supply voltages, so a new method needs to be developed [59].

Until the deep submicron era, transistor characteristics scaled predictably with respect to transistor dimensions and supply voltage. This was quantified into generalized scaling equations that took short-channel effects into consideration [60, 61]. Great gains were generated following this method, with few outside factors, designers could simply use smaller transistors for added performance [58]. These scaling factors have been used and taught, so they are easily found in the literature [56, 57], and are shown in Table 2.1 where scaling factor S is the ratio of the transistor dimensions between two transistor sizes and U is the ratio between two voltages. With both S and U, it is expected that all geometry and voltages scale together.

As transistors get smaller, however, short-channel effects and other issues such as process variation start playing a larger role, making the traditional scaling equations inaccurate [8, 55, 58, 62]. Leakage current is affected greatly by gate length, oxide thickness, and threshold voltage, so it is becoming a large issue with deep submicron processes, where these values are small, and getting smaller. With these issues affecting transistor operation, designers started looking to optimize between technology nodes other than simple geometric scaling. Width, length, and oxide thickness are not scaling together, and neither is supply voltage, V_{DD}, and threshold voltage, V_T, which means that scaling factors S and U shown in Table 2.1 can not be determined. The above mentioned non-regularities were especially noticeable when the industry switched largely to using high-k dielectrics and metal gates with technology nodes at 45 nm and smaller [63] and again when the industry switched to multi-gate (double gate/FinFET or tri-gate) transistors at 20 nm and smaller [8, 58, 64, 65]. This will of course be further complicated in the not so distant future beyond CMOS, when it becomes commercially viable to use different devices for continued performance gains,

Table 2.1: Traditional scaling equations for short-channel devices. Adapted from Rabaey [56].

Parameter	Relation	Full Scaling	General Scaling	Fixed V Scaling
W, L, t_{ox}		$1/S$	$1/S$	$1/S$
V_{DD}, V_T		$1/S$	$1/U$	1
$Area/Device$	WL	$1/S^2$	$1/S$	$1/S^2$
$Power$	$I_{sat}V$	$1/S^2$	$1/U^2$	1
$IntrinsicDelay$	$R_{on}C_G$	$1/S$	$1/S$	$1/S$
$Energy$	Pt	$1/S^3$	$1/SU^2$	$1/S$

such as nano-electro-mechanical (NEM) devices [6], carbon nanotube transistors [7], or nanowire transistors [8, 9]. The presented modeling method could potentially be used to characterize scaling to these devices, but is not covered here.

Method for Accurate Scaling in Deep Submicron Technologies

The physics of transistor operations in the submicron region become far more complicated than those in the micron region, with leakage and other above mentioned issues becoming large factors in energy consumption and delay. The most accurate way to get scaling factors in submicron processes is to use a Spice simulation tool, such as HSpice with a model that specifies the characteristics of the particular technology. Simulating a whole design in Spice with modified technology size and voltages would result in the most accurate comparison [56]. While this is an accurate method, it is impossible without the complete extracted design, which makes this an unviable option by which to compare multiple competitive designs. This work presents factors for estimated performance scaling between technology nodes. There is a lack of applicable methods in the literature to predict CMOS circuit design performance in deep submicron technology as it scales between different technology nodes to present and near future technologies without extracted netlists from the target CMOS circuit design.

This work expands upon a preliminary report [66] by using Spice simulation results to model CMOS device performance from different technology nodes to create scaling factors of energy,

delay, power, and area between nodes and supply voltages. One of the large motivations for this project was to create the ability to compare different simple digital hardware implementations using a fair metric. A good performance approximation of a device in a certain technology can be achieved using inverters in a chain, with 4 inverters attached to each output, this is known as Fan Out 4, or FO4. A circuit that has a delay and consumption of X number of FO4 inverter chains in a certain technology size should have roughly the same X number of FO4 inverter chains in a different technology size [67]. With this in mind, this work sets out to take simulated measurements of FO4 models in a range of different sizes and voltages to obtain approximated scaling factors for power, energy, and delay. These factors can be used to scale performance measurements when comparing digital designs with different fabrication technologies and supply voltages.

2.3.1 Background

International Technology Roadmap for Semiconductors (ITRS)

The International Technology Roadmap for Semiconductors (ITRS) [68] creates reports that predict where semiconductor technology is headed in the next 15 years. These reports are formed by a collaboration of many companies and research institutions. In this work, these reports were used to obtain industry standard technology sizes, and voltages commonly used, as well as general knowledge about transistor changes over the years. Area is also of interest in digital design, so this work evaluates minimum feature sizes, 1 half Metal 1 pitches, and 4 transistor (4T) logic gate sizes as scaling factors. In this report, when technology process node sizes are referred to, i.e. 180 nm, 45 nm, etc., it is referring to the minimum feature size. Process sizes were generally identified by their smallest feature size, and for a long time, DRAM 1/2 pitch sizes were the smallest, and were therefore used to identify technologies. With new fabrications, this has not been the case, and ITRS discontinued identifying technologies by their minimum feature size. To try to stop confusion, they started to differentiate by using the first year of production. However, in their 2013 report, they started giving "node name" labels to more easily correlate to industry terminology [68]. As minimum feature technology nodes have continued to be the generally accepted term, in this work technologies are identified by both the production year and technology node, as shown in Table 2.2.

Table 2.2: Characteristics of different technology nodes [68]. The modeled measurements are for a single minimum sized inverter in an FO4 chain. The energy value is the average energy required for a single inverter transition from low to high, or high to low.

				Simulated Performance of Inverter		
Production Year	Technology Node (nm)	Technology Type	V_{DD} (V)	Delay (ps)	Energy (fJ)	Power (µW)
1999	180	Bulk	1.8	77.2	27.5	105
2001	130	Bulk	1.2	34.7	5.20	26.1
2004	90	Bulk	1.1	26.5	2.62	13.0
2007	65	Bulk	1.1	19.8	1.72	8.58
2008	45	High-k	1.1	10.9	1.05	5.19
2010	32	High-k	0.97	9.8	0.51	2.47
2012	20	Multi-Gate	0.9	9.66	0.198	1.51
2013	16*	Multi-Gate	0.86	6.12	0.179	1.28
2013	14*	Multi-Gate	0.86	4.02	0.144	0.995
2015	10	Multi-Gate	0.83	3.24	0.122	0.866
2017	7	Multi-Gate	0.8	2.47	0.111	0.789

* The 2013 ITRS report labels a single "16/14" node.

Predictive Technology Model (PTM)

The Predictive Technology Models [65, 69, 70, 71, 72], or PTMs, were used to simulate different performance characteristics as technology size and voltage scaled. The models were developed for designers who do not have access to proprietary transistor characteristics to test designs with future technology nodes. The PTMs are the most accurate models available, as semiconductor companies do not readily provide characteristics of their specific technologies. This lack of specificity of PTM has the added benefit of generality for our purpose of comparing designs

across fabrication technologies and manufacturers.

2.3.2 HSpice Device Modeling

HSpice was used to model the scaling results. A fan out four, FO4, inverter chain was used. FO4 delay has been shown to be proportional to CV/I (intrinsic capacitance, voltage, and drive current of a device) [67]. Intrinsic capacitance and drive current are both proportional to device size, so they scale with $1/S$ and as previously mentioned, voltage scales at factor U, thus, using traditional scaling methods, delay should scale with U/S^2.

The inverters in the model were designed as a 4× minimum size CMOS inverter for the technology node, with a PMOS to NMOS ratio of $\beta = 2$ to keep the rise and fall time roughly balanced. In a multi-gate transistor, the effective channel width is equal to two times the height of the fin plus the width of the fin, or $W_{eff} = 2 \times h_{fin} + W_{fin}$ [65]. After the effective channel width of a single fin was determined, the number of fins in the transistor was modified to achieve a 4× minimum size CMOS inverter with a PMOS/NMOS ratio of $\beta = 2$.

The modeled inverter chain starts with one inverter with the output connected to 4 identical inverters, with that output connected to 16 inverters, and so on until the circuit ends with 64 inverters, creating a total of 4 FO4 stages. A square wave was modeled as the input to the inverter chain. The set of 16 inverters in the middle of the chain, were used for the sampling. The delay between when the input signal to the set of 16 inverters crossed the midpoint and the output crossed the midpoint was measured. The voltage was measured along with the current, and calculations were made by equations 2.1-2.4 where t_0 to t_1 is the transition time as the signal transitions from 10% V_{DD} to 90% V_{DD}, and t_2 to t_3 is the transition time from 90% V_{DD} to 10% V_{DD}.

$$P_{ave} = \frac{1}{T} \int_0^T I(t) \cdot V dt \tag{2.1}$$

$$E_{0\to1} = \int_{t_0}^{t_1} I(t) \cdot V dt \tag{2.2}$$

$$E_{1\to0} = \int_{t_2}^{t_3} I(t) \cdot V dt \tag{2.3}$$

$$E_{ave} = \frac{E_{0\to1} + E_{1\to0}}{2} \tag{2.4}$$

Simulation

The simulations were run on the following technology sizes: 180 nm, 90 nm, 65 nm, 45 nm, 32 nm, 20 nm, 16 nm, 14 nm, 10 nm, and 7 nm with supply voltages varying from 1.8 V to 0.5 V in 0.05 V increments. Technology nodes are not designed to handle voltages much higher or lower than their target voltages, so even though HSpice gave results for the technology nodes operating at non-expected voltages, they were removed from the results as the PTM characteristics are not expected to hold for these values.

With the industry standard of high-k dielectric transistors at 45 nm and below, high-k PTM models, both for high performance (HP) and low power (LP), are used for the 45 nm and 32 nm nodes. High performance transistors are generally designed with lower threshold voltages, which allows for faster switching times, at the expense of leakage power. Low power transistors are generally the opposite, with high threshold voltages, which gives lower power consumption, especially while in standby, with low leakage. Also, as the industry standard for 20 nm and below is multi-gate transistors, the PTM models for both HP and low standby power (LSTP) are evaluated for these devices between 20 nm and 7 nm. Low standby power multi-gate transistors, similar to low power transistors, target lower power, at the cost of propagation delay.

As transistors become smaller, wire load is making a larger impact on total device performance [56]. However, this is affected by many things, many of which are fabrication or design specific, such as how long a signal must travel on a wire, the number of contact vias between metal layers and their size, or wire dimensions. It was determined to be impractical to include a factor that can have so much undeterminable variance, so wire loads were not included. For larger technologies, and smaller designs, this will affect the factor less but it should be considered when using these scaling factors.

2.3.3 Simulation Results and Scaling Factors

Table 2.2 shows the standard values labeled by ITRS at each technology node investigated, as well as the delay, energy, and power simulated using the inverter chain in HSpice as described in Section 2.3.2. The V_{DD} is taken from the ITRS tables for high-performance.

Table 2.3: Geometric sizes of different technology nodes which affect area from ITRS reports [68].

Minimum Feature	Metal I Half	(4T) Logic Gate
Size (nm)	Pitch (nm)	Size (μm^2)
180	230	57
130	150	10.4
90	90	5.2
65	68	2.6
45	59	2.1
32	45	0.71
20	32	0.35
16/14	40	0.248
10	31.8	0.157
7	25.3	0.099

Area

To determine a factor for scaling area between technologies, minimum feature sizes, Metal 1 half pitches, and 4T logic gate of MPU (High-volume Microprocessor) were taken from the ITRS reports. An exact scaling would be dependent on the design, but using either of the aforementioned values should give a good estimate, especially for simple designs. The relative scaling of area is shown in Figure 2.3 and detailed in Table 2.3. The geometric characteristics with a single length dimension were squared to get an area value. To combine all of these factors for a useable average, the geometric mean was taken to give each of the factors an equal weight. Table 2.4 displays the scaling factors using the geometric mean of the area factors presented in Table 2.3. To scale area, determine the scaling factor by finding the intersection of the starting technology node and desired technology node row. Multiply that number by the starting chip's area to determine an equivalent area in the desired technology node. The closest scaling to the traditional scaling factors in Table 2.1

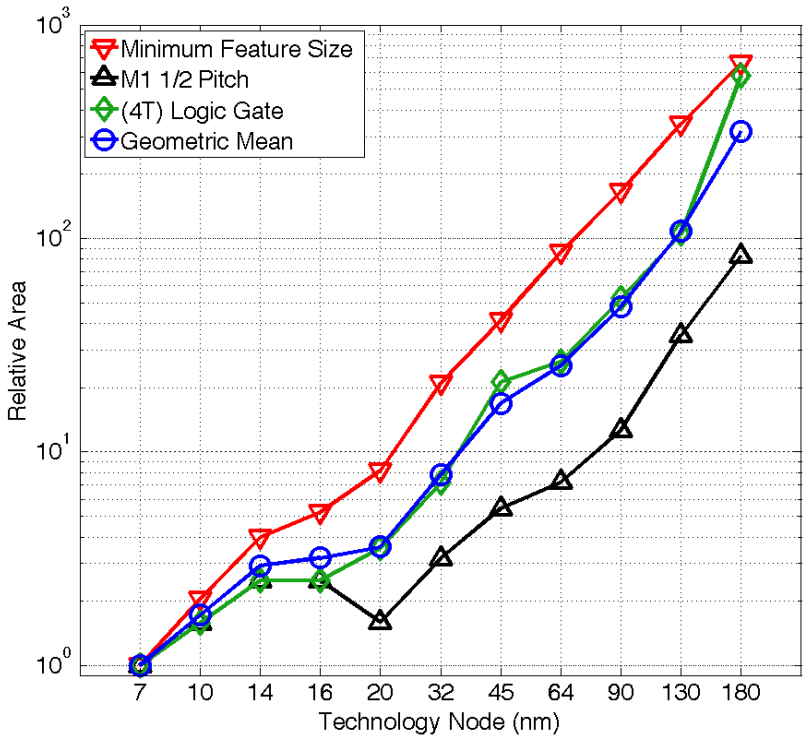


Figure 2.3: Relative area scaling of different area sizes over different technology nodes, and the geometric mean of all three of the sizes.

would be for a scaling factor S using the minimum feature size.

Delay, Energy, and Power Scaling Factors

The results from the HSpice simulation of the inverter chain are given in Figures 2.4, 2.5, and 2.6. Figure 2.4 plots the average propagation time of a single inverter in the middle of an FO4 inverter chain. Figure 2.5 plots the average energy required for a state change of this single inverter. Figure 2.6 plots the average power consumption of the single inverter over an entire 1000 ps clock period. This takes into account the increased leakage of the smaller technology nodes.

To compare against traditional scaling methods, Figures 2.7, 2.8, and 2.9 use the nominal supply voltage values given in Table 2.2 to plot the modeled data of the major technology nodes. Using the traditional scaling equations in Table 2.1, the traditional scaling methods are used both scaling from the 180 nm node and the 7 nm node to the other technology nodes. The traditional methods fail by a considerable margin in all but scaling power values from 7 nm, as shown in Figure 2.9.

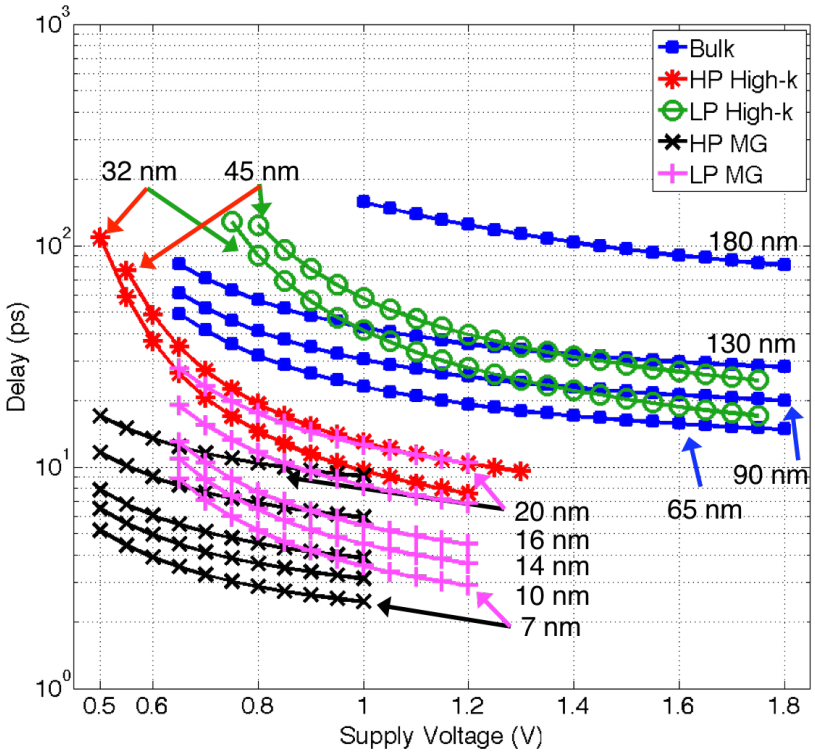


Figure 2.4: Delay for a signal to propagate through one inverter in the middle of the FO4 inverter chain for different technology nodes with scaling voltage.

As printing all of the data points from the multitude of HSpice simulations would be prohibitive, polynomial approximations for each of the performance factors, i.e. the modeled delay, energy, and power associated with a particular technology node and supply voltage, are generated for ease of use, without loss of accuracy. The polynomial approximations were created using a script that iteratively increased the order of the polynomial until a coefficient of determination, or R^2, value of greater than 0.95 was attained. This resulted in a third-order polynomial for the delay factor approximations, and second-order polynomials for energy and power factor approximations.

Equations 2.5, 2.6, and 2.7 are used to determine a $DelayFactor$, $EnergyFactor$, and $PowerFactor$, respectively, for a specific technology node and voltage.

$$DelayFactor = a_{d3}V^3 + a_{d2}V^2 + a_{d1}V + a_{d0} \tag{2.5}$$

$$EnergyFactor = a_{e2}V^2 + a_{e1}V + a_{e0} \tag{2.6}$$

$$PowerFactor = a_{p2}V^2 + a_{p1}V + a_{p0} \tag{2.7}$$

The coefficients for the above equations corresponding to each technology node can be found in

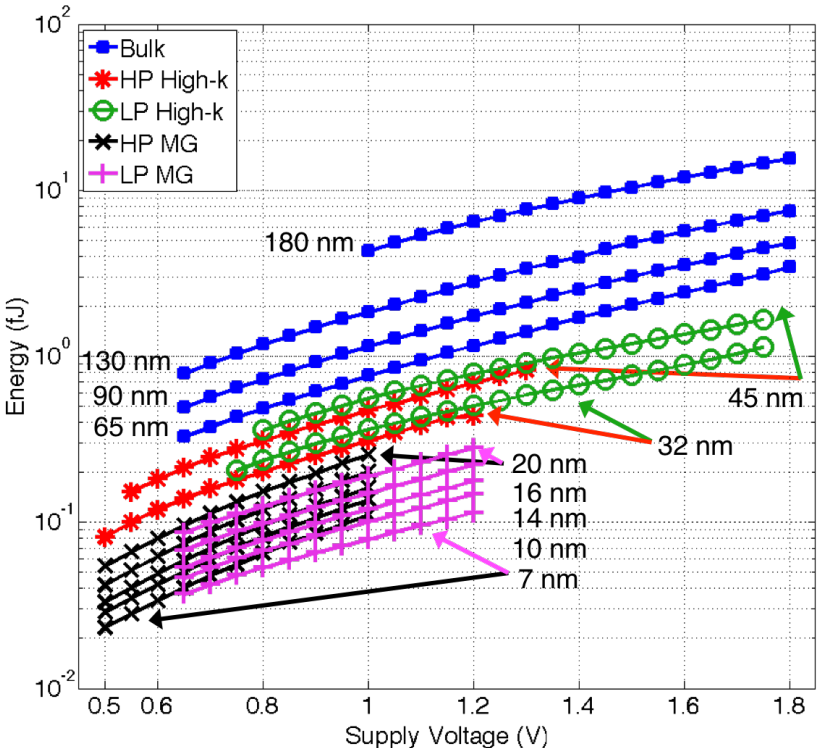


Figure 2.5: Energy required for one inverter in the middle of the FO4 inverter chain to toggle for different technology nodes with scaling voltage.

Table 2.5. For simplicity, all coefficients were rounded to four significant figures, which did not significantly effect the coefficient of determination. This level of accuracy is more than sufficient given the imprecision inherent in technology scaling without full design knowledge. It is not recommended to use these factors for supply voltages without a corresponding data point in Figures 2.4, 2.5, and 2.6 for a particular technology node, as those voltages are outside of the normally operating voltages of that particular technology node.

Equations 2.8, 2.9, and 2.10 may be used to scale delay, energy, and power, respectively, between technology nodes.

$$D_x = \frac{DelayFactor_x}{DelayFactor_y} \cdot D_y \tag{2.8}$$

$$E_x = \frac{EnergyFactor_x}{EnergyFactor_y} \cdot E_y \tag{2.9}$$

$$P_x = \frac{PowerFactor_x}{PowerFactor_y} \cdot P_y \tag{2.10}$$

As the modeled power values are an average over a 1000 ps clock period, it largely displays standby power for each node. For a more accurate power scaling factor for a high-performance circuit at

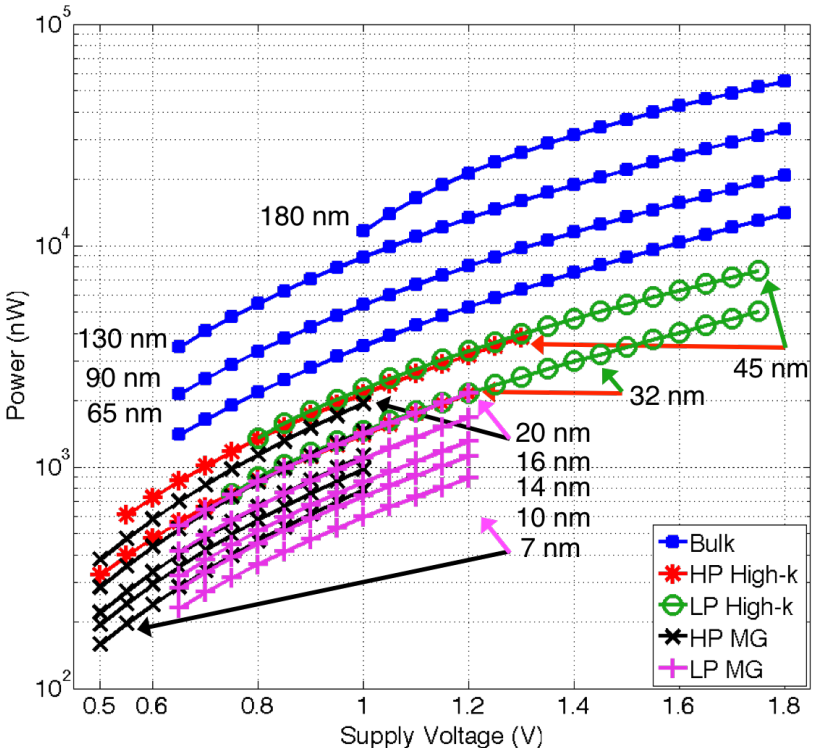


Figure 2.6: Average power signal for a clock cycle in which a signal is propogated through one inverter in the middle of the FO4 inverter chain for different technology nodes with scaling voltage.

maximum speed, Equation 2.11 may be used.

$$P_x = \frac{EnergyFactor_x \cdot DelayFactor_y}{EnergyFactor_y \cdot DelayFactor_x} \cdot P_y \tag{2.11}$$

In Equations 2.8–2.11, subscript x refers to the desired node, while y refers to the starting node. The factors $DelayFactor$, $EnergyFactor$, and $PowerFactor$ are obtained from Equations 2.5, 2.6, and 2.7, respectively. If this paper is viewed online, one can alternatively use the interactive plot to attain values for $DelayFactor$, $EnergyFactor$, and $PowerFactor$ to be used in Equations 2.8–2.11.

Scaling Example

The following example is given to illustrate the scaling procedure. To scale 1 pJ/Op from 1.3 V in 65 nm to 0.9 V in HP 32 nm, Equations 2.6 and 2.9 are used, as shown in Equations 2.12, 2.13,

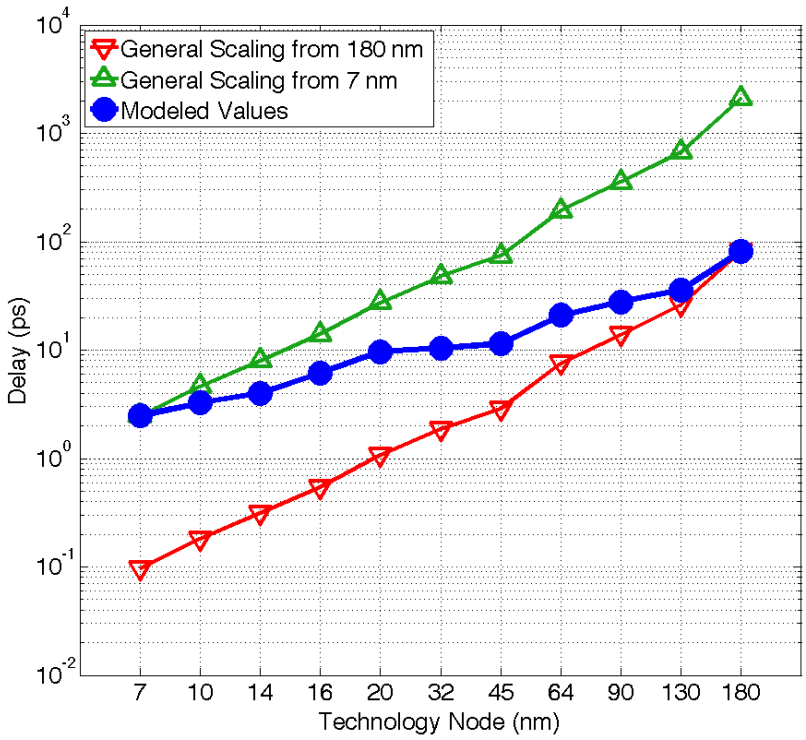


Figure 2.7: Delay simulated for a signal to propagate through one inverter in the middle of the FO4 inverter chain using Table 2.2 values and scaled using Table 2.1 equations.

and 2.14.

$$EnergyFactor_x = a_{e2}V^2 + a_{e1}V + a_{e0}$$

(from Equation 2.6)

$$EnergyFactor_x = 0.5654(0.9)^2 - 0.2962(0.9) + 0.1148 \tag{2.12}$$

(from Table 2.5)

$$EnergyFactor_x = 0.3062$$

$$EnergyFactor_y = a_{e2}V^2 + a_{e1}V + a_{e0}$$

(from Equation 2.6)

$$EnergyFactor_y = 2.441(1.3)^2 - 2.831(1.3) + 1.276 \tag{2.13}$$

(from Table 2.5)

$$EnergyFactor_y = 1.721$$

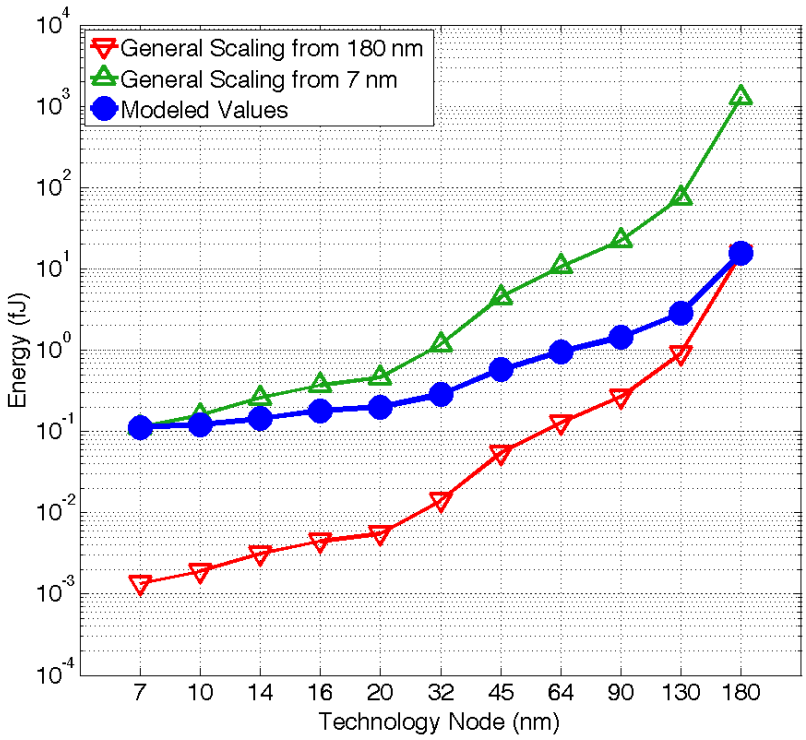


Figure 2.8: Energy used to toggle one inverter in the middle of the FO4 inverter chain simulated using Table 2.2 values and scaled using Table 2.1 equations.

$$E_x = \frac{EnergyFactor_x}{EnergyFactor_y} \cdot E_y \text{ (from Equation 2.9)}$$
$$E_x = \frac{0.3062}{1.721} \cdot 1\,\text{pJ/Op} \tag{2.14}$$
(from Equations 2.12 and 2.13)
$$E_x = \mathbf{0.1779\ pJ/Op}$$

This work presents a method and data sets from simulation that can be used to scale transistors to different technology nodes in a fair method [59]. The data presented shows that traditional scaling methods do not hold into these submicron transistors, especially with the advent of radically changed devices. The general trend is similar, but would not make an accurate comparison. Thus the method of using the modeled simulation data presented in this work is a more accurate estimation that can be used to compare two devices from different technologies and supply voltages.

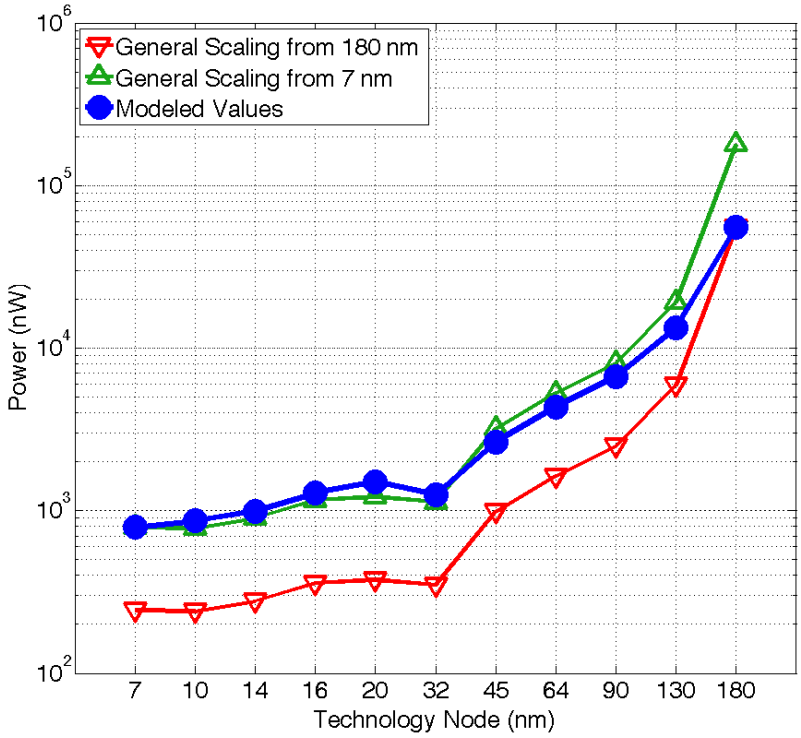


Figure 2.9: Average power for a clock cycle simulated for a signal to propagate through one inverter in the middle of the FO4 inverter chain using Table 2.2 values and scaled using Table 2.1 equations.

Table 2.4: Area scaling factors using geometric mean of area values given by the three sizes from Table 2.3.

Desired Node \ Starting Node	180 nm	130 nm	90 nm	65 nm	45 nm	32 nm	20 nm	16 nm	14 nm	10 nm	7 nm
180 nm	1	0.34	0.15	0.08	0.053	0.025	0.011	0.01	0.0093	0.0055	0.0032
130 nm	2.9	1	0.44	0.23	0.16	0.072	0.033	0.03	0.027	0.016	0.0092
90 nm	6.6	2.3	1	0.53	0.35	0.16	0.075	0.067	0.061	0.036	0.021
65 nm	12	4.3	1.9	1	0.66	0.31	0.14	0.13	0.12	0.068	0.039
45 nm	19	6.4	2.8	1.5	1	0.46	0.21	0.19	0.17	0.1	0.059
32 nm	40	14	6.1	3.3	2.2	1	0.46	0.41	0.38	0.22	0.13
20 nm	88	30	13	7.1	4.7	2.2	1	0.89	0.82	0.48	0.28
16 nm	99	34	15	7.9	5.3	2.4	1.1	1	0.91	0.54	0.31
14 nm	110	37	16	8.7	5.8	2.7	1.2	1.1	1	0.59	0.34
10 nm	180	63	28	15	9.8	4.5	2.1	1.9	1.7	1	0.58
7 nm	320	110	48	25	17	7.8	3.6	3.2	2.9	1.7	1

Table 2.5: The polynomial coefficient values to be used with Equations 2.5-2.7 to attain the factors to be used to generate the scaling factor between two technology nodes and voltages.

Type		Node	Delay Coefficients (Eq. 2.5)				Energy Coefficients (Eq. 2.6)			Power Coefficients (Eq. 2.7)		
			a_{d3}	a_{d2}	a_{d1}	a_{d0}	a_{e2}	a_{e1}	a_{e0}	a_{p2}	a_{p1}	a_{p0}
Bulk		180 nm	-	97.09	-356.7	406.5	-	24.64	-17.98	-	101000	-79720
		130 nm	-76.65	334.9	-493.4	275.8	7.171	-6.709	2.904	27020	-15450	5630
		90 nm	-60.34	262.5	-384.2	210.9	4.762	-4.781	2.092	17320	-11230	4328
		65 nm	-53.3	230.4	-333.9	178.6	3.755	-4.398	1.975	12890	-10510	4362
High-k	HP	45 nm	-501.6	1567	-1619	566.1	1.018	-0.3107	0.1539	5462	-1760	522.4
		32 nm	-1047	2982	-2797	873.5	0.8367	-0.4341	0.1701	4001	-1733	533.6
	LP	45 nm	-285.7	1239	-1795	898.8	1.103	-0.362	0.2767	6297	-3009	1124
		32 nm	-325.9	1374	-1922	913.2	0.9559	-0.7823	0.471	4557	-3037	1323
Multi-Gate	HP	20 nm	-	34.63	-66.37	41.15	0.373	-0.1582	0.04104	2922	-1286	299.9
		16 nm	-	24.8	-47.52	28.87	0.2958	-0.1241	0.03024	2133	-882.6	197.7
		14 nm	-40.66	109.2	-100.6	35.92	0.2363	-0.09675	0.02239	1675	-711	159
		10 nm	-34.95	93.65	-85.99	30.4	0.2068	-0.09311	0.02375	1456	-621.6	143.8
		7 nm	-28.58	76.6	-70.26	24.69	0.1776	-0.09097	0.02447	1179	-515.7	123.4
	LSTP	20 nm	-160.5	514.1	-558.6	217.5	0.2632	-0.14	0.06841	2096	-962.4	287.1
		16 nm	-114.6	366.7	-397.4	153.6	0.2139	-0.1187	0.05639	1609	-715.5	205.7
		14 nm	-85.37	271.6	-292.2	111.4	0.1556	-0.06472	0.03066	1259	-554.1	152.3
		10 nm	-71.76	228.6	-246.3	93.91	0.1261	-0.0518	0.02769	1046	-422.7	118.9
		7 nm	-61.79	196.1	-210.3	79.55	0.09365	-0.03409	0.02043	815.2	-307.3	87.54

2.4 Many-Core Processor Array Design

Modern semiconductor fabrication technologies now enable the construction of integrated circuits containing over 1000 processors on a single chip. However, for such a system to effectively compute workloads, new architectures are needed for the processors themselves, the interconnect that binds them, subsystems that interact with larger memories, and the applications that they execute.

Throughout the history of computing, adding functionality onto a single chip has virtually always brought increased performance at a reduced cost [1]. While it is a near-certainty that more chips containing 1000 processors will be built in the future [4, 12], there are challenging open questions regarding how those processors will communicate with each other and external system components, and how applications will be written for them. The KiloCore and KiloCore2 chips demonstrate the feasibility and some advantages of this promising new era.

There has been work into hardware design specifically for such a large many-core system, such as hybrid floating-point modules [73], allowing partial implementation of the normally large floating-point units to maintain small individual core area, which allows for applications required floating point to perform accurate computations if reduced precision floating point is able to be performed, such as synthetic aperture radar (SAR) image formation [74, 75]. There has also been work on application specific hardware or accelerators for such systems, such as a H.265 compliant motion estimator [76].

For decades, virtually all processors have contained memory caches, which hold a small subset of frequently used data and instructions. While they are extremely effective for most programs, they are problematic for resources such as chip input and output, and cache-coherency circuits when the number of processors is scaled into the 100s or 1000s [2, 77]. Caches also dissipate substantial power, typically in the range of half of the processor's total power consumption [2]. In contrast, KiloCore and KiloCore2 processors do not contain caches and instead store data and instructions inside i) local memory, ii) an arbitrary number of nearby processors, iii) independent on-chip memory blocks, or iv) off-chip memory, which can be managed by flexible programmer-specified software if desired when large amounts of data or program storage are needed.

A many-core processing array must be most effective working with computationally-

intensive workloads which are those where energy dissipated processing data dominates energy dissipated by mass storage or large-scale networks. It is typically possible to find tasks or kernels in these workloads that either fit into one, 10s, 100s, or more processors, or can be structured so that large amounts of data can be processed by passing through clusters of processors. Many-core processor arrays will find a place in systems both as a standalone processor and as a companion to a host processor which would handle less-intensive and large-memory software. Many applications have been implemented on a large many-core array, helping support the desire for such a product. The platform many of these applications were implemented on, AsAP2 [78], was originally created for digital signal processing (DSP), and many DSP applications have been successfully implemented, such as advanced encryption standard (AES) processing [79, 80], a H.264 encoder [81, 82], and a LDPC decoder [83], all of which showed high throughput and energy efficiency. It has also been expanded to other database applications, such as sorting [46, 84, 85] and string search [86], which show the promising possibilities of integrating a large energy-efficient many-core system into a larger datacenter.

Virtually all digital integrated circuits are clocked by global or semi-global clocks generated by extremely stable but high-power phase-locked loops (PLLs) synchronized to a crystal oscillator reference [87]. In contrast, in a GALS processing array, processors and memory blocks are clocked by individual, local, completely-unconstrained (below the maximum operating frequency) clock oscillators that do not use PLLs and may change frequency, halt, and restart within a few clock cycles at will to reduce power dissipation. Processors 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 spreading complex software workloads evenly over thousands of processors which leads to the increasing prevalence of processors that are not always active. Many-core processor arrays can allow for a near-perfect proportional scaling of power dissipation and processor workloads.

Chapter 3

Sorting on a Many-Core Array

3.1 Proposed Sorting Kernels

The sorting applications, described in Section 3.2, utilize basic program kernels in each of the processors in the array. Each kernel was designed for modularity with no knowledge about a processor's location or temporary list size. Each processor has a 128x35-bit instruction memory, limiting each kernel to just 128 instructions.

3.1.1 SAISort Kernel

The serial array of insertion sorts, or SAISort, is the fundamental sorting block used in all variations of the presented sorts. The name is taken from the observation that the sort on the micro scale is an insertion sort, and when multiple kernels are serially linked together, a bubble sort is created in the macro scale.

As shown in Algorithm 1, when a fresh list is sent through the SAISort kernel, it starts by filling up its internal memory with a sorted list, as records arrive. Our target architecture has a local memory of 256 bytes, so only two 100-byte records can be locally stored at a time. Once the local memory is full, each new record is sorted into the local list and the lowest valued record is sent out. A code word preceding each record is reserved for triggering a reset at the end of a temporary list, allowing temporary lists to begin without foreknowledge of the list's size. When the kernel detects a reset signal, it passes the reset signal to its output and begin the record output streaming protocol described in Section 3.2.4.

Algorithm 1 SAISort Kernel

```
RecCount ← 0
while true do
    if Input ≠ Reset then
        if RecCount < 2 then
            SortedList[RecCount] ← InputRec
            if RecCount == 0 then
                RecCount ← 1
            else if SortedList[0].Key ≤ SortedList[1].Key then
                RecCount ← 2
            else
                RecCount ← 3
            end if
        else
            if InputRec.Key ≤ SortedList[RecCount − 2].Key then
                Output ← InputRecord
            else
                Output ← SortedList[RecCount − 2]
                SortedList[RecCount − 2] ← InputRec
                if SortedList[0].Key ≤ SortedList[1].Key then
                    RecCount ← 2
                else
                    RecCount ← 3
                end if
            end if
        end if
    else
        Output Procedure (See Section 3.2.4)
        RecCount ← 0
    end if
end while
```

Algorithm 2 Merge Kernel

```
while true do
    ResetFlag ← 0
    while ResetFlag == 0 do
        if Input1Rec.Key < Input2Rec.Key then
            Output ← Input1Rec
            if Input1 == Reset then
                ResetFlag ← 1
            end if
        else
            Output ← Input2Rec
            if Input2 == Reset then
                ResetFlag ← 2
            end if
        end if
    end while
    if ResetFlag == 1 then
        while Input2 ≠ Reset do
            Output ← Input2Rec
        end while
    else
        while Input1 ≠ Reset do
            Output ← Input1Rec
        end while
    end if
end while
```

3.1.2 Merge Kernel

The merge kernel, shown in Algorithm 2, compares the keys received on each of its two inputs and passes the record with a lower key value to its output. When a reset signal a received from one input, records from the other input are passed directly to the output until a second reset signal is received. The kernel passes the reset signal and then resumes merging.

Algorithm 3 Split Kernel

```
while true do
    CountPass ← 0
    while Input ≠ Reset do
        if CountPass ≠ Ratio then
            OutputPass ← InputRec
            CountPass++
        else
            OutputRow ← InputRec
            CountPass ← 0
        end if
    end while
    OutputPass ← Reset
    OutputRow ← Reset
end while
```

3.1.3 Split Kernel

The split kernel, shown in Algorithm 3, is given a constant value at compile time for the number of split cores past the current core. The kernel passes that many records to the next split processor, then takes one record for its current row. It continues in this fashion until it receives the reset signal, at which point it forwards the reset signal to both the next split processor and the current row, then start from the beginning of the program.

3.1.4 Distribution Kernel

The distribution kernel, shown in Algorithms 4, 5, and 6, presorts records by the MSBs of their key and sends them to rows of the array for finer sorting. When a reset is triggered, records are streamed one row at a time in increasing radix order. The radix value used by each core is set at compile time.

Three rows of cores are combined into a single lane that processes two radixes and is managed by a single distribution core. The upper and lower rows are each dedicated to a radix, while the middle row processors are dynamically assigned to the upper or lower row as those rows fill. Control signals for the middle row configuration are generated by the distribution core for a

Algorithm 4 Radix Distribution

```
while true do
    if ThisCore == FirstCore then
        if AutoResponseMode + PipedResponseReq == 0 then
            if RecordsPending ≥ PipeMaxCountDiv2 then
                DwnStrOutput ← ResponseRequest
                PipedResponseRequests++
            end if
        else
            if RecordsPending ≥ PipeMaxCount then
                Process Response (See Algorithm 5)
            else
                Process Packet (See Algorithm 6)
            end if
        end if
    else
        if UpStreamInput/neq Empty then
            Process Response (See Algorithm 5)
        else if DwnStrInput/neq Empty then
            Process Record (See Algorithm 6)
        end if
    end if
end while
```

lane. The sorting cores use the SAISort kernel.

A series of distribution cores handle the routing of records to the appropriate lane and row, as well as detecting when a row is full and triggering a reset. When a record enters a distribution core, its key is compared to the core's unique threshold value, and passes the record to the next distribution core if above the threshold or is processed to be sent into a row otherwise, as shown in Algorithm 6. These thresholds progressively increase along the series of distribution cores such that a random key has an approximately equal chance to be sent to any row.

The rate at which records are admitted into the array is controlled by the first distribution core, using information sent from other distribution cores to avoid overflowing any lane. Records

Algorithm 5 Process Response

```
if DwnStrInput == Reset then
    (UpStream, UpRow, LowRow)Output ← Reset
    Perform Reset
end if
if ThisCore == FirstCore then
    RecordsPending− = InputAccountedRecordsTotal
    if RecordsPending ≤ AutoResponseCuttoff then
        AutoResponseMode ← 1
        DwnStrOutput ← AutoResponseEnableSignal
    end if
    PipeMaxCount ← InputCapacity
    PipeMaxCountDiv2 ← InputCapacity >> 1
else
    MinCap ← MIN(UpRoom, LoRoom, DwnStrCap)
    UpStreamOutput ← MinCap
    UpStreamOutput ← InputAccountedRecordsTotal + AccountedRecords
    AccountedRecords ← 0
end if
```

are streamed in a two stage algorithm, as described below.

During the first stage, records are streamed from the first distribution core up to a limit of the minimum remaining capacity of any row. With each record passed, a counter for records pending response is incremented. When the count has passed half of the target limit, a request is sent out for updated information. Distribution cores pass the request downstream if they have passed records downstream since the previous response, else they respond to the request. The response consists of the minimum capacity of either sorting row in this lane or the minimum of the next upstream core, and a count of the number of records processed since the last response was sent. Both of these values are updated by each core passing the response.

When the first distribution core reaches a number of records pending response equal to its stored minimum remaining capacity, it reads the response to update this capacity and reduce the counter for records pending response. Records resume streaming using this updated information.

Algorithm 6 Process Record

if $UpStreamInput ==$ Reset **then**

$(DwnStr, UpRow, LowRow)Outputs \leftarrow$ Reset

Perform Reset

else if $UpStreamInput ==$ ResponseRequestSignal **then**

if $RecordPassedSinceLastResponse == 1$ **then**

$RecordPassedSinceLastResponse \leftarrow 0$

$DwnStrOutput \leftarrow$ ResponseRequestSignal

else

$MinCap \leftarrow MIN(UpRoom, LoRoom, DwnStrCap)$

$UpStreamOutput \leftarrow MinCap$

$UpStreamOutput \leftarrow AccountedRecords$

$AccountedRecords \leftarrow 0$

end if

else if $UpStreamInput ==$ AutoResponseEnableSignal **then**

$AutoResponseMode \leftarrow 1$

$DwnStrOutput \leftarrow$ AutoResponseEnableSignal

else if $Key >$ CuttoffValue **then**

$RecordsPassedSinceLastResponse \leftarrow 1$

$DwnStrOutput \leftarrow Record$

if ThisCore == FirstCore **then**

$RecordsPending+ = 1$

end if

else

if ThisCore $\neq$ FirstCore **then**

$AccountedRecords+ = 1$

end if

if $Key \leq LowerRowCuttoffValue$ **then**

$SelectedRow \leftarrow$LowerRow

$OtherRow \leftarrow$UpperRow

else

(Continued on next page)

(Algorithm 6, continued from previous page)

$SelectedRow \leftarrow$UpperRow

$OtherRow \leftarrow$LowerRow

end if

if $SelectedRowRoom \neq 0$ **then**

$SelectedRowRoom--$

if $AutoResponseMode == 1$ **then**

$MinCap \leftarrow MIN(UpRoom, LoRoom, DwnStrCap)$

$UpStreamOutput \leftarrow MinCap$

$UpStreamOutput \leftarrow 1$

end if

$SelectedRowOutput \leftarrow Record$

else

if $SelectedRowUnassignedNodes \neq 0$ **then**

if $SelectedRow == UpperRow$ **then**

$UpperRowOutput \leftarrow$ ReconfigurationSignal

$LowerRowOutput \leftarrow$ ReconfigurationSignal

end if

$SelectedRowRoom+ = 7$

$OtherRowRoom+ = 4$

$SelectedRowUnassignedNodes- = 1$

if $AutoResponseMode == 1$ **then**

$MinCap \leftarrow MIN(UpRoom, LoRoom,$

$DwnStrRoom)$

$UpStreamOutput \leftarrow MinCap$

$UpStreamOutput \leftarrow 1$

end if

(Continued on next page)

(Algorithm 6, continued from previous page)

SelectedRowOutput ← *Record*

else

AllOutput ← Reset

Reprocess Latest Record

end if

end if

end if

When the safe record limit is sufficiently large, the first distribution core may continue to stream records during the entire time the requested response is being processed and returned.

The second stage of distribution is entered when the minimum safe limit falls below a threshold value, set to approximately the number of distribution cores for optimization. A signal is sent downstream through the distribution cores indicating the change in stage. During this stage, whenever a distribution core processes a packet it automatically sends a response upstream containing an updated minimum capacity. The first distribution core progressively checks these responses and admit additional records into the array as it verifies there is guaranteed remaining capacity for them.

Sorting of a set of records ends when a lane receives a record and determines it should be placed into a row that is already full. The core sends a reset request to the upstream and downstream distribution cores, as well as both of its lanes sorting rows. The core then resets its state and reprocess the record that triggered the reset. Other distribution cores propagates the reset signal and reset. A reset signal may also be sent into the core array to force a reset.

The advantage of stage one is that it avoids excessive processing of responses, reducing the cycles needed per record and filling the array quickly when room is plentiful. The advantage of stage two is that it provides much finer grained response arrival to the first distribution core when the ability to hide response latency is limited by the low minimum safe remaining capacity of one or more lanes.

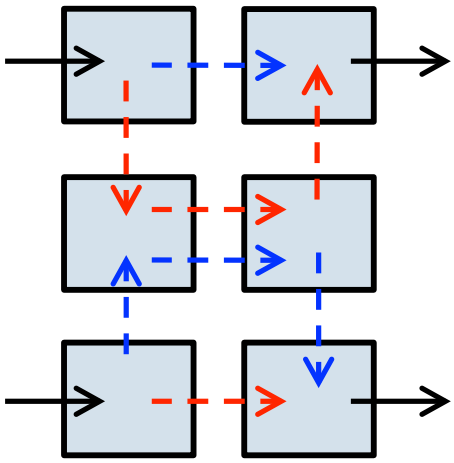


Figure 3.1: A block of 6 Adaptive Sorting processors route data to dynamically change the data path. Either the shown blue or red routing is taken depending on need. These 6 processors constitute a part of a single lane, which contains two sorting rows.

3.1.5 Dynamic Routing

Assignment of the center SAISort processors in a lane to the upper or lower rows is performed in 3x2 blocks, with two cores in each row as shown in Fig. 3.1. Upon reset, a block defaults to assigning the central cores to the lower row; upon reception of a reconfiguration signal from the distribution core, the central cores are reassigned to the upper row. The reassignment is performed by reconfiguration of the circuit network control signals in each core of the block, allowing records to be diverted from the upper or lower row into the central cores for additional capacity. To prevent reconfiguration of the block when it is still active, the central cores sends a signal to the upper and lower rows to indicate when they have completed a flush and are ready for reset. The upper and lower rows delay reconfiguration until the signal is received.

3.2 Proposed Sorting Applications

Several different sorting schemes were explored. The following sorting applications fit within our target architecture limitations, utilize simple modular kernels, and scale easily with processor array sizes.

3.2.1 Snake Sort

The snake sort is the simplest of the proposed sorting variations. This sort uses the SAISort kernel on each processor in the array, linking them together using a single input and output per processor. Each processor will take an input record, determine where it fits in that processor's sorted list, and will then output the lowest record. To fit within the processor memory limitation of 256 bytes on our target architecture, each processor core can hold up to two 100-byte records. Each processor added to the array increases the number of records sorted per temporary list by two.

One final record can be added to the temporary list, giving the temporary list size Equation 3.1 where n is the number of processors in the array and R is the number of records that can fit inside a single processor's local memory.

$$TempListSize = n \times R + 1 \tag{3.1}$$

Any additional records added will be sorted incorrectly if they have a value lower than that of a record previously pushed out of the array. After a full temporary list has been sent into the chip, a reset signal is sent through the snake triggering a flush in each processor. The output protocol is covered in Section 3.2.4.

A generalized mapping for the Snake Sort is shown in Fig. 3.2 which displays how more processors are used.

3.2.2 Row Sort

In row sort, multiple lists are sorted in parallel and then merged together, shortening the data path for any given record. When data enters the processor array, split processors in the first column are used to evenly distribute the records to each of the rows in the array. Each row utilizes the SAISort to sort their given records similar to individual snake sorts, generating multiple sorted

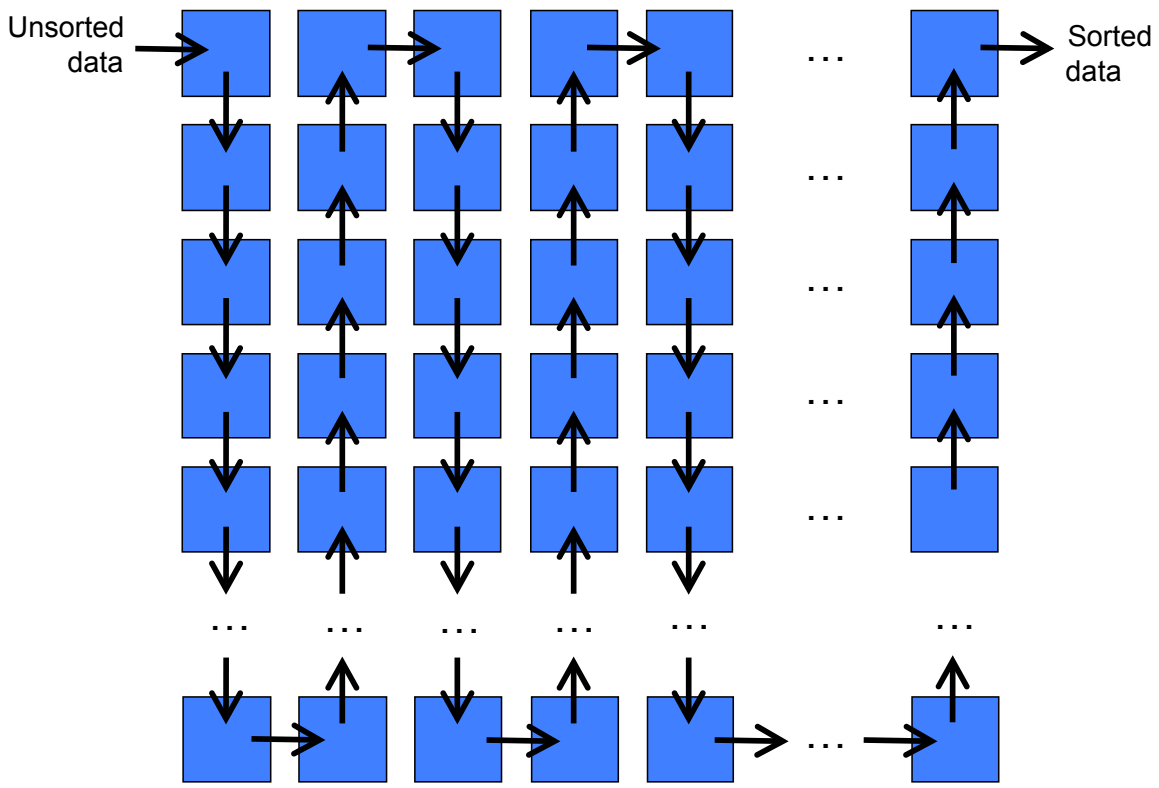


Figure 3.2: An example snake sort mapping showing the communication path and how it can be scaled.

lists. When the reset signal is sent into the chip, each row will send its sorted list to the final column of processors, where the merge kernel will join the lists to output a single sorted list. A mapping of the row sort is given in Fig. 3.3 where the sort can be scaled by adding more rows or columns.

Due to overhead for the split and merge kernels, this method sorts fewer records per temporary list than snake sort. The total number of records in a temporary list can be given by Equation 3.2 where r is the number of rows, c is the number of columns, and R is the number of records that can fit in a single processor's local memory.

$$TempListSize = (r \times c - 2 \times (r - 1)) \times R + r \tag{3.2}$$

The number of sorting processors is given by the total number of processors, r times c, minus $r - 1$ distribution and $r - 1$ merge processors. The bottom left and upper right processors are not required for distribution or merging respectively, as shown in Fig. 3.3. Two records are stored per sorting core and an additional record is added to each row, similar to snake sort. It was found that the highest activity processors with this method was the merge column of processors.

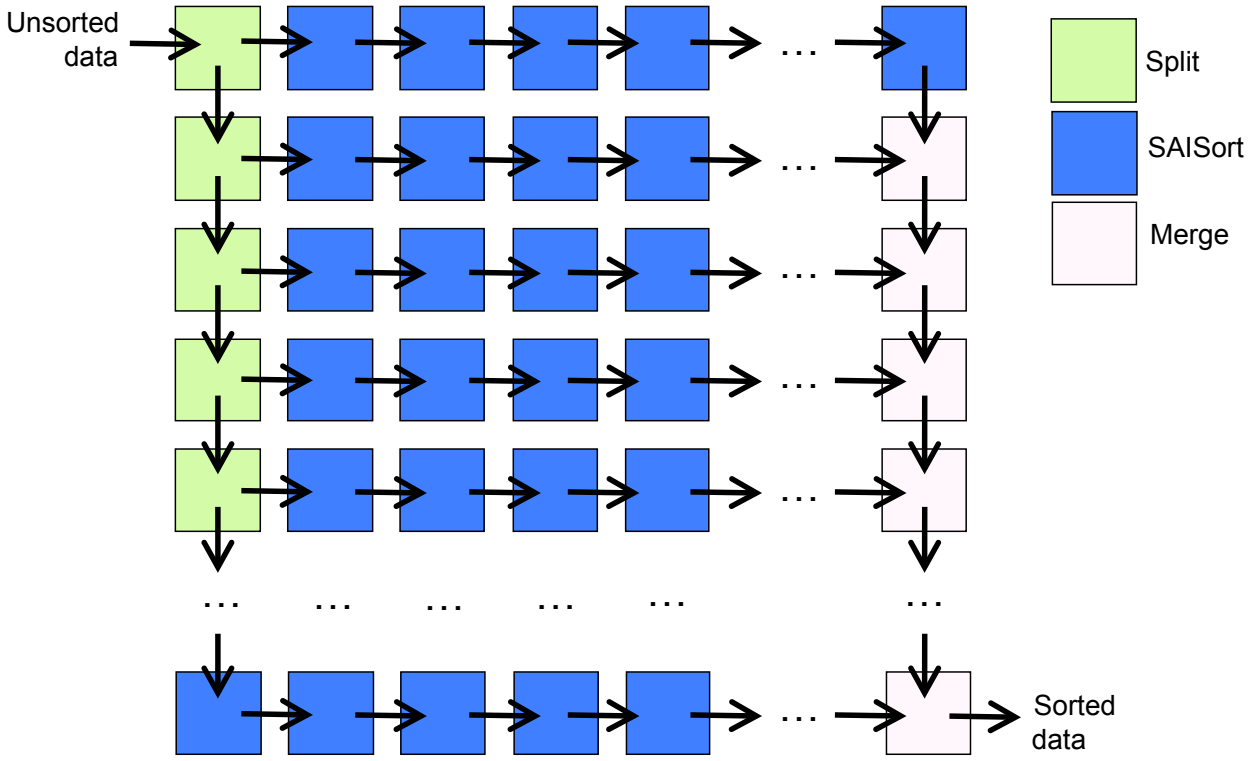


Figure 3.3: An example row sort mapping showing the communication path and how it is scaled.

3.2.3 Adaptive Sort

In adaptive sort, the processor array is partitioned into a number of sorting lanes. Each lane begins with one 3x1 block containing one distribution processor and two basic SAISort processors. The lane is then expanded with some number of 3x2 adaptive sorting blocks, shown in Fig. 3.1, each containing the SAISort kernel with additional reconfiguration code, and connected as shown in Fig. 3.4. Each lane is responsible for sorting two rows of radixes.

The distribution cores of each lane are connected together for passing records and control signals. The final sorting cores for each radix are similarly connected together for emptying records from the array. The lane connected to chip input runs additional distribution code for controlling the rate records are admitted into the array.

The advantage of radix sort over row sort is that it avoids a merging bottleneck at the end of each temporary list. The disadvantages are a more complicated distribution algorithm and a

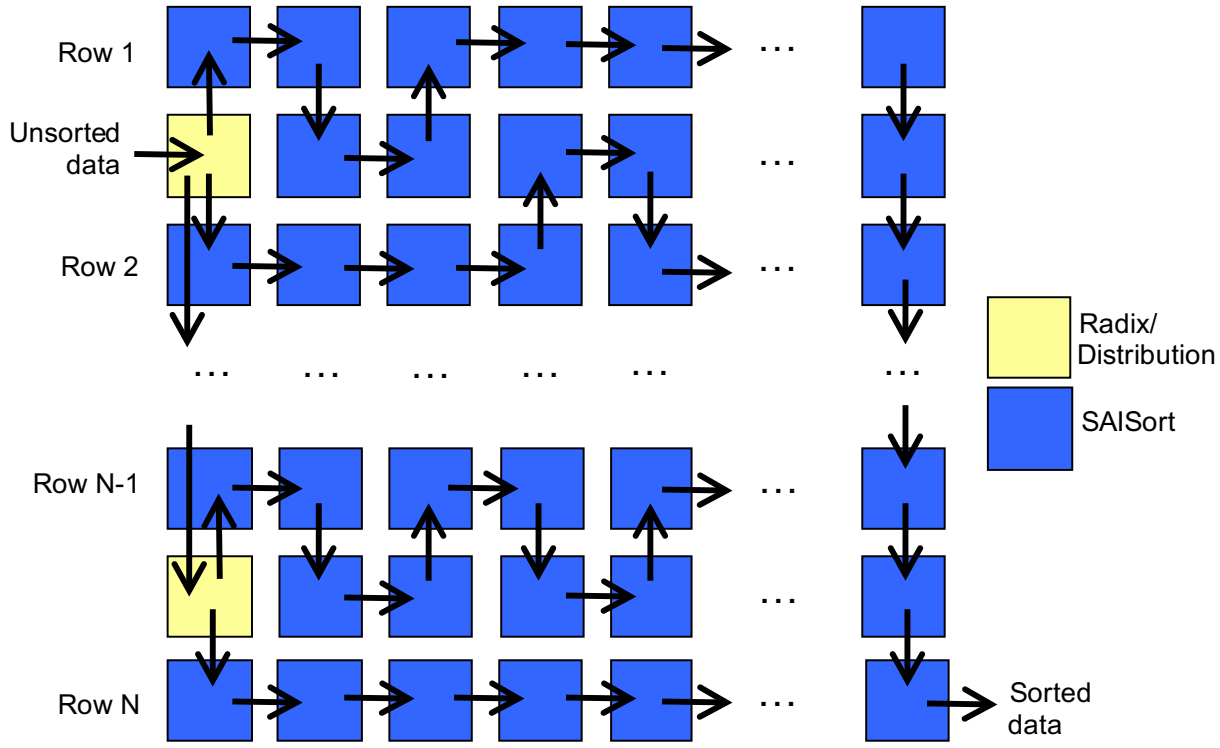


Figure 3.4: An example adaptive sort mapping showing the communication path and how it is scaled. There are two radix rows assigned to every three rows of processors, where the central processors are dynamically allocated during runtime. In this example rows 1 and 2 filled up at a similar rate so they were allocated a similar number of middle processors, but row N-1 filled up faster than row N, so all of the middle processors were dynamically allocated to row N-1.

non-deterministic number of records per temporary list, as each run ends when a radix is full.

The maximum potential temporary list size is given Equation 3.3 where L is the number of sorting lanes, b is the number of adaptive 3x2 blocks per lane, and R is the number of records that can fit in one processor's local memory. Similar to snake sort, one extra record can be sorted for each radix, or potentially two per lane.

$$TempListSize = L \times R \times (2 + 6 \times b) \tag{3.3}$$

3.2.4 Two Output Streaming Protocols

When the chip contains a completely sorted temporary list, it then needs to stream that data out of the array. Two such protocols were explored.

Algorithm 7 Flush

```
NumToPass ← Input
Output ← Reset
if RecCount == 2 then
    Output ← NumToPass + 2
    Output ← SortedList[0 : 1]
else if RecCount == 3 then
    Output ← NumToPass + 2
    Output ← SortedList[1]
    Output ← SortedList[0]
else if RecCount == 1 then
    Output ← NumToPass + 1
    Output ← SortedList[0]
end if
for i = 1 to NumToPass do
    Output ← InputRec
end for
return
```

Flush

With this protocol, each processor will receive a count of how many records are held by upstream processors, output its two local stored records, and subsequently, directly pass the upstream records from input to output. While simple, the flush protocol requires the first sorting processor to hold its stored records until all downstream processors have passed their own stored records, delaying the beginning of a new temporary list. The delay would be even worse if the buffers used for inter-processor communication weren't large enough to take an entire record. The pseudo code is shown in Algorithm 7.

Pass and Store

This protocol is similar to flush except that input records are temporarily stored in a processor's local memory before being passed to output as shown in Algorithm 8. The processor will continually send out its lowest record and then store a new record into the vacated memory.

When all upstream records have been read, the processor will output its final stored records. Pass and store effectively compresses a temporary list into the downstream processors while waiting for records to exit the chip, allowing the upstream processors to begin a new temporary list almost immediately. Compared to flush, the drawbacks of pass and store include increased code size, more than doubled operations per record, and additional memory reads and writes.

Algorithm 8 Pass and Store

```
NumToPass ← Input
Output ← Reset
if RecCount < 2 then
    if RecCount == 1 then
        Output ← NumToPass + 1
        Output ← SortedList[0]
    else
        Output ← NumToPass
    end if
    for i = 1 to NumToPass do
        Output ← InputRec
    end for
    return
else if RecCount == 2 then
    Output ← NumToPass + 2
else
    Output ← NumToPass + 2
    if NumToPass ≥ 1 then
        Output ← SortedList[1]
        SortedList[1] ← InputRec
        NumToPass − −
    end if
end if
for i = 1 to NumToPass do
    if i rem 2 == 1 then
        Output ← SortedList[0]
        SortedList[0] ← InputRec
    else
```

(Continued on next page)

(Algorithm 8, continued from previous page)

$Output \leftarrow SortedList[1]$

$SortedList[1] \leftarrow InputRec$

end if

end for

if $NumToPass$ **rem** $2 == 1$ **then**

$Output \leftarrow SortedList[1]$

$Output \leftarrow SortedList[0]$

else

$Output \leftarrow SortedList[0:1]$

end if

return

3.3 Experimental Methodology

All of the proposed sorts follow the Sort Benchmark [88] requirement of 100-byte records, which is composed of a uniformly random 10-byte key and a 90-byte payload. As the Sort Benchmark only uses random data, no other types of data distributions are explored. The gensort program [89] was utilized to create random datasets for all experiments.

The most relevant Sorting Benchmark to the proposed work is the JouleSort [32], which measures the total system energy cost of sorting a data set. This paper proposes using a many-core array as a co-processor in a database system, freeing the general purpose CPU to process other workloads during the first phase of sorting. Therefore, it would be uninformative to try and add a system power to our sorting results, as required by the JouleSort, so we focus on just the processing energy, which is not reported for other sorts. There is no way to compare our first phase results to other results, as metrics separated by phase are rarely published, and even if they were, they would not match the size of our temporary lists, which would make any comparison uninformative.

The proposed many-core sorting results are compared with an Intel Core2 Duo T7700 and an Nvidia GeForce 9600m because no published results are available for a wide variety of block sizes such as the ones in Table 3.2, and because hardware and software tools are widely available for them. In addition, although the design team sizes and resources are vastly different for the three platforms, they are all built from the same 65 nm fabrication technology generation which results in a very fair comparison in this critical parameter. Technology scaling trends can give a first order prediction of how a device or ASIC would perform in a different technology [59, 66], however, as smaller transistors allow for a higher transistor density, other architectural and system design considerations affect large scale chip and processor performance between technology nodes [56]. Therefore, scaling the performance of an external sort on a general-purpose processor to a different technology node is not possible. This makes it impossible to compare fairly with hardware from a different technology. Table 3.2 also contains a comparison with a JouleSort winner which used a 65 nm Intel Core2 T7600 processor, the CoolSort [32].

3.3.1 Many-Core Sorts

The scalable many-core architecture is modeled using a cycle-based, two-state simulator written in C++ which is able to model arbitrarily sized arrays. Each processor core is simulated as a separate task, allowing multithreaded operation. Core simulations are synchronized based on the timing of data transfers through a circuit network. Energy usage is estimated based on physical chip measurements for each type of operation, memory accesses, network access, oscillator activity, and leakage.

Each of the proposed sorts are implemented on many-core arrays with the number of processors scaled from 10 to 10,000 processors with a clock speed of 1.2 GHz at 1.3 V. Phase 1 results are shown in Section 3.5.1. Some selected points for the size of the array are used to compare full external sorts in Section 3.5.2.

3.3.2 Intel CPU Quick Sort

The quick sort [90] was chosen because it is widely considered among the most efficient sorts. It was implemented in C++ on an Intel Core2 Duo T7700, a 65 nm chip with a TDP of 35 W and a clock frequency of 2.4 GHz. The implementation saturated the thread count of the processor, to make sure to take advantage of both processing cores. The main memory was preloaded with records for multiple temporary lists of a size matched to the many-core sorts. Thousands of runs were timed to calculate the throughput. The program was compiled with the gcc compiler and optimization set to –O3. Similar to the many-core sorts, little effort was put into hand optimizing this sort.

3.3.3 Nvidia GPU Radix Sort

The radix sort was chosen for the GPU platform because there was a version publicly available with the Nvidia CUDA SDK version 2.3, written by Satish et al. [49]. The sort was implemented on an Nvidia GeForce 9600m GT, a 65 nm chip with 32 processing elements, a clock speed of 500 MHz, and a power consumption of 23 W. The older CUDA SDK version was chosen for its compatibility with our CUDA compute 1.1 GPU. The original sort was designed to efficiently work with the 32 bit data width, so it only sorted keys that were 32 bits or smaller. Therefore it

was necessary to modify the code to work with the 80-bit keys and 100-byte total records. The main memory of the graphics card was preloaded with records for multiple temporary lists in sizes matching the inputs for the many-core sorts before the timing was started. Thousands of runs were processed through the platform to get an average throughput result.

Because of the SIMD nature of the GPU processing array, if a dataset does not completely pack the array, excess processing elements will have wasted cycles. This can be reduced with large temporary list sizes, but with some of the smaller temporary list sizes tested, this became a factor causing sub-optimal results.

After the GPU used in this comparison was created, Nvidia developed GPUs which have architectural advancements targeted at GPGPU programming, as well as a toolkit which is not backwards compatible and contains a sort that is more efficient because it uses these new advances. While a sort could have ran faster on a newer chip, it would have created an unfair comparison as the 65 nm many-core chip was not afforded extra architectural changes or smaller transistors.

3.3.4 Intel CPU Merge Sort

In order to simulate an entire external sort, it was necessary to write a sort for phase 2. The merge sort was chosen due to it's simplicity [33]. This was implemented in C++ on the same Intel Core2 Duo processor described in Section 3.3.2. One GB of sorted lists were loaded onto main memory and timed separately from the sort where thousands of records were merged to get a throughput number. As large arrays of hard drives are used to negate the access and read/write times, we used this assumption in our models after determining our bandwidth was comfortably under the theoretical bandwidth of 68 Gb/s in the previous JouleSort winner, FAWNSort's setup [91]. The throughput number was used to model the total time to complete phase two for the different temporary list sizes.

3.4 Many-Core First Phase Sort Analysis

The following section presents sorting approaches and algorithms for the first phase of an external sort, which are targeted to operate on a low power fine-grained many-core processor array [46]. This work explores only the first phase, and uses the AsAP2 processor to quantify the

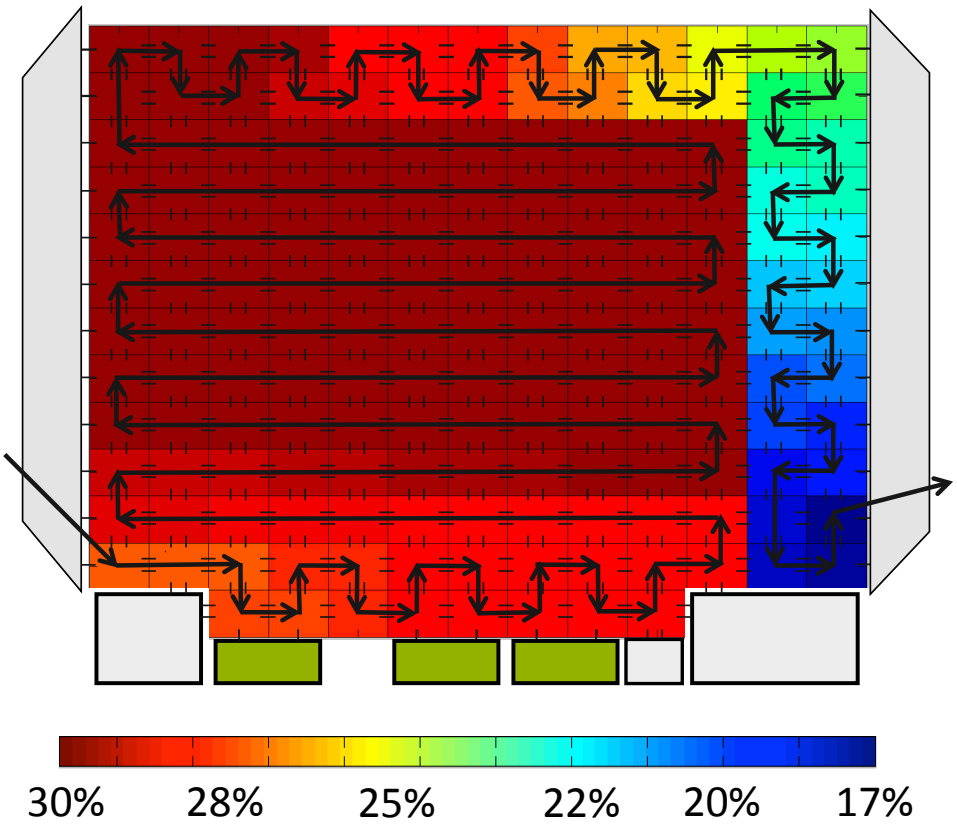


Figure 3.5: "Heatmap" showing processor activity levels while executing the snake sort.

results. This preliminary work only explored the snake and row sorts, with only the flush output method.

3.4.1 Processor Activity Percentage

The percentage of time that each processor is actively processing information varies among the different processors in the array, highlighting bottlenecks in the sorts. Figure 3.5 shows the activity percentage of each processor when one run is sent through the snake sort. The first processors' lack of activity would be filled up by successive runs when running sorts on a large set of data, allowing runs to overlap. The processors are not fully utilized because they are stalled waiting for a new input or waiting to output a record. The activity level decreases starting at around the 120^{th} core reaching an activity percentage near 17% at the last processor in the snake. These last processors see a reduced amount of utilization because they pass a majority of the list through without needing to sort the data.

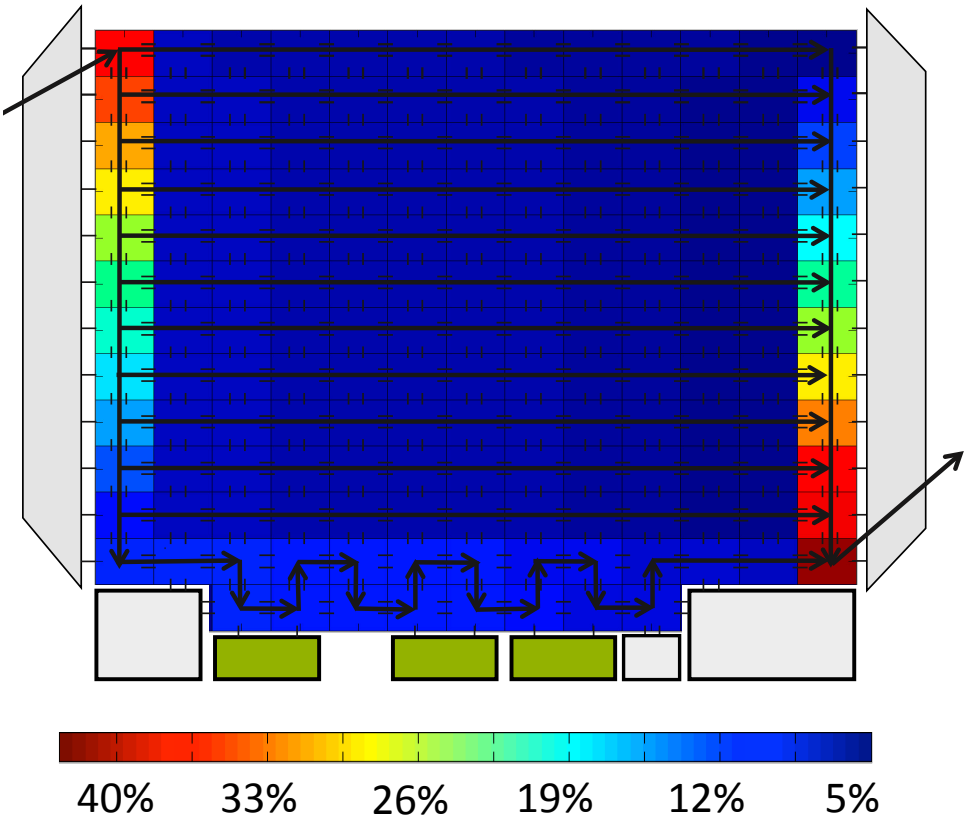


Figure 3.6: "Heatmap" showing processor activity levels while executing the row sort highlighting the bottleneck at the end, and the low power consumption in the middle.

Figure 3.6 shows the activity percentage of the row sort while sorting one run of data. The merging processors are between 7% to 41% active. The 41% active merge processor at the bottom of that column is the bottleneck, as every entry needs to be merged through this one processor, while the other merging processors only work on subsets of the run. The distribution processors in Figure 3.6 are stalled waiting for more input records. With multiple runs of data, the activity of the distribution processors and the sorting processors would increase, again showing overlap that exists between runs.

The activity percentages are all below 100% utilization since they wait for neighboring processors, which decreases the throughput. However, as each processor in the array can completely halt its oscillator, the stalling of the processors has a negligible affect on the total energy consumed.

3.4.2 Experimental Results

A version of the proposed many-core sort was run on a physical AsAP2 chip. Unfortunately, the laboratory setup does not have a method to stream data into the processor, as is required by the test conditions. The version of the sorts run on the physical chip contain a random record-generating program on the first processor on the chip, which has a lower throughput than what is required to avoid stalling the sorting processors. This occurs since at the beginning of each run, each processor will take in two records to fill its memory, which requires data to be input as fast as it can be transmitted. Thus, any timing numbers would be invalid. Therefore, all timing measurements are obtained from a cycle-accurate Verilog simulator, running the same Verilog code that was used to create the chip. It is possible to connect input and output to the chip with a different setup and could easily be implemented as a co-processor on a database system. All of the power numbers are calculated in the Verilog simulator, using measured power numbers for different operations. The metrics of throughput (rec/sec), throughput per area ((rec/sec)/mm^2), and energy dissipated per record (nJ/rec), are used. The chip die size of each platform was used to calculate the throughput per area.

To get comparable results using the comparison sorts described in Section 3.3 are executed with thousands of runs with the same number of records per run as those of the proposed sorts were run through the program to record an average throughput. The program fills up main memory, and then a timer is started. The sorts are performed, and then the timer is stopped before the results are written to secondary memory, in an effort to remove system I/O speeds from the results. The random data was created by the gensort program [89]. The results of the Intel sorts were scaled from 45 nm to the 65 nm technology node to match the AsAP2 and Nvidia platforms [66]. Due to the simplicity and modular format of the many-core sorting versions, programing each of these comparison sorts took more time then programing each of the AsAP sort versions.

The throughput and energy efficiency for the 100 byte record sorts are shown in Table 3.1.

Figure 3.7 compares the presented work with the sorts ran on the Intel and Nvidia platforms, with energy efficiency on the vertical axis and throughput on the horizontal axis. The AsAP2 processor operated with a clock speed of 1.07 GHz at 1.2 V and a clock speed of 260 MHz at 0.75 V. The highest throughput is over twice the number of records per second and over twenty seven

Table 3.1: Throughput and energy dissipation for sorting 100-byte data records on a single temporary list of varying sizes.

Records Per Block	Processor		Throughput (1000 rec/sec)	Throughput per Area ((rec/sec)/mm^2)	Energy per Record (nJ/rec)
296	Intel Core 2 Duo quicksort		1,300	8,100	6,700
	Nvidia 9600M GT radixsort		1,500	10,000	15,000
	AsAP2, Row Sort	1.2 V	**8,900**	**220,000**	90.8
		0.75 V	2,200	55,000	**19.8**
329	Intel Core 2 Duo quicksort		1,300	8,100	6,700
	Nvidia 9600M GT radixsort		1,500	10,000	15,000
	AsAP2, Snake Sort	1.2 V	4,400	110,000	670
		0.75 V	1,070	27,000	146
753	Intel Core 2 Duo quicksort		1,200	7,500	7,200
	Nvidia 9600M GT radixsort		1,500	10,000	15,000
	AsAP2, Row Sort	1.2 V	6,600	170,000	108
	w/ On-chip Memories	0.75 V	1,600	41,000	25.6
785	Intel Core 2 Duo quicksort		1,200	7,500	7,200
	Nvidia 9600M GT radixsort		1,500	10,000	15,000
	AsAP2, Snake Sort	1.2 V	3,800	95,000	646
	w/ On-chip Memories	0.75 V	910	23,000	143

times the number of records per second per mm^2 higher than the sort on the Intel processor. From Table 3.1, it can be seen that the AsAP2 throughput is higher than the Intel sort's, and it does this at less than half the clock speed of the Intel processor. The most energy efficient sort is 330 times more efficient than the sort on the Intel processor.

Table 3.1 shows that the fastest AsAP2 sort throughput is over twenty-two times higher than the GPU radixsort and the lowest energy consuming AsAP2 sort is over 750 times more efficient than the the Nvidia GPU sort. It should be noted that to make the GPU sort comparable to this work, it was necessary to sort a data set that is suboptimal for the SIMD architecture of the GPU.

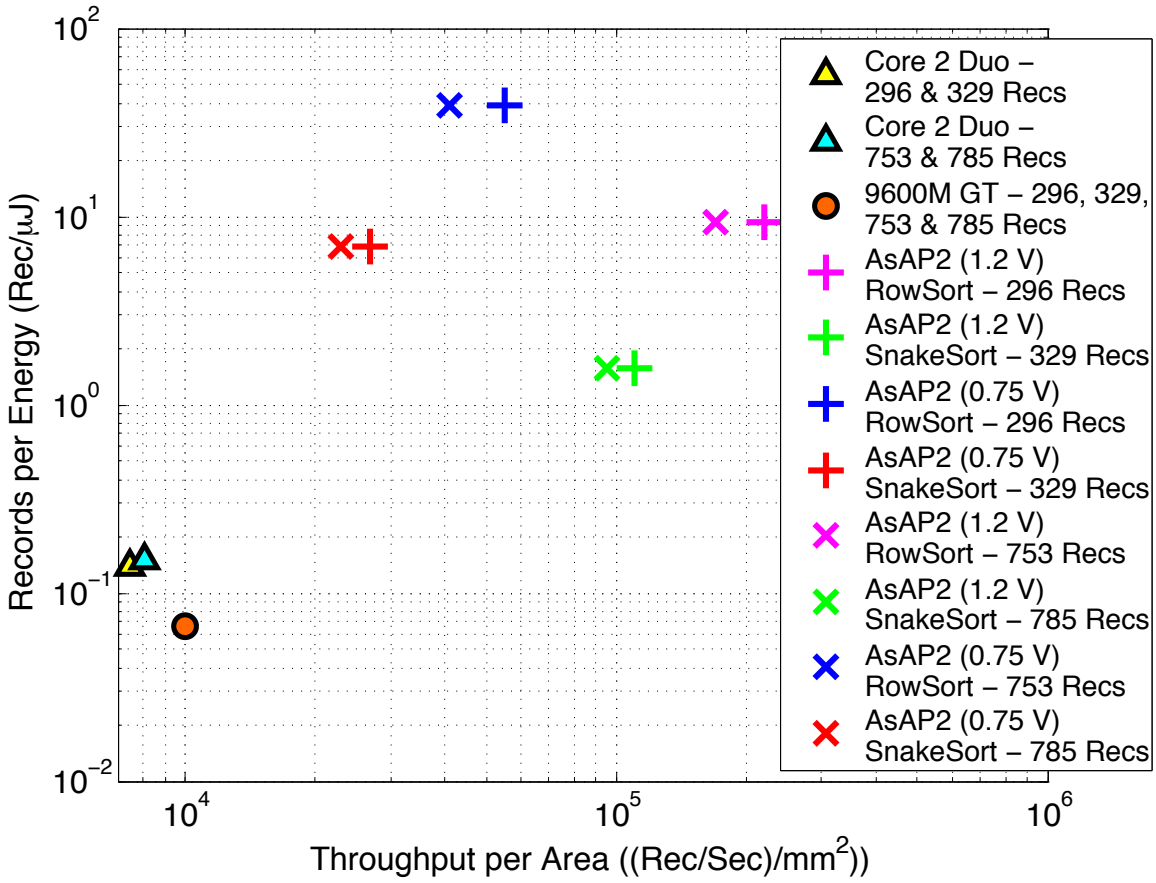


Figure 3.7: Comparison of sorting 100-byte records on AsAP2, a Nvidia 9600M GT, and a Core 2 Duo, scaled to 65 nm.

The GPU would preform best with all all 512 threads in a block full, so run sizes of either multiples of 512 or sufficiently large so that the blocks could be packed for multiple runs would have been more efficient. To change the run size to be optimal for the GPU would have given incomparable results to this work. In Table 3.1 it can be seen that all of the throughputs are the same for the GPU, this again can be attributed to the SIMD architecture. With similar run sizes, the number of blocks run in the CUDA program were nearly identical, meaning the effort for the GPU was about the same for all of the sorts with varying amounts of empty threads running.

Figure 3.7 clearly indicates that the row sort has the highest throughput and lowest energy usage per record. It can also be seen that adding the memories increases the size of the run without a significant change to throughput or energy usage per record. The snake sort recognizes a small decrease in energy used per record, which occurs as the sorting is shifted to fewer merge processors. The row sort sees a small increase in energy per record as it already uses series of of merges, The

largest difference in the sorts which used the memories are the costs associated with powering the on chip memories, and merging more records.

3.5 Simulated Results of 10 GB External Sort

Expanding on previous work, the next step was to expand the sorting for a general many-core platform [85]. This models an entire external sort of 10 GB with varying sized arrays from 100 to 10,000. Building on previous work [46, 84], a new sorting method, adaptive sort, was introduced, as well as the store and pass method of outputting data.

3.5.1 Phase 1 Processor Scaling Results

The three proposed many-core sort applications were simulated using between 10 and 10,000 processors in a single square array.

Figure 3.8 shows the energy usage per record for the different implementations. The snake sort energy usage is considerably greater than the row sort consumption and to a lesser extent, the adaptive sort consumption. The snake sort has a greater rate of increase of energy consumption, where the other two have a diminishing rate of increased energy consumption. This happens because with the snake sort every record must be operated on by every processor in the sort, where the other two sorts divide the workload. With the row and adaptive sorts, the individual workload of a SAISort processor only goes up with each additional processor added to a row, and would go down with the addition of another row. The pass and store output protocol was able to smooth out the throughput of the row sort because it removed the penalty of the random records clogging a row, and slowing down the sort.

Figure 3.9 shows throughput for snake sort, row sort, and adaptive sort using both pass and store and flush output protocols. The throughput is held relatively steady as the number of processors and corresponding list size grows. Pass and store shows higher throughput than flush for snake and row sorts but slightly reduced throughput for adaptive sort. In the latter case, often only one sorting row will be full at the end of a sorting a temporary lists; other rows will have room to move records downstream and free upstream processors for a new temporary list, mitigating the benefit of pass and store. This benefit is then overwhelmed by the increased operations per record

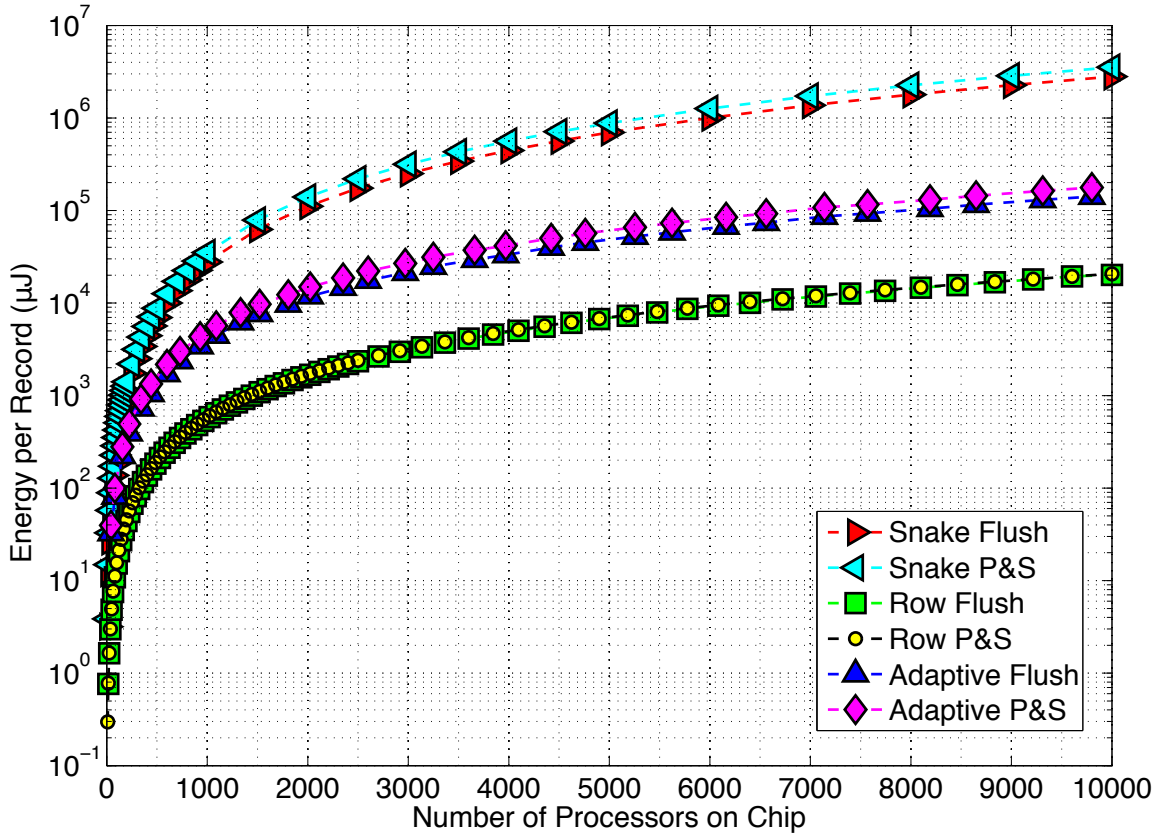


Figure 3.8: Energy per record used by the many-core array while scaling the number of processors with the phase 1 of the proposed sorts. With each new processor, the temporary list size increases, as described in Equations 3.1–3.3.

for the pass and store protocol, requiring more time for records to exit the upstream processors and delaying a new temporary list.

The row sort is the most energy-efficient and has the highest throughput, when compared to the other sorts. The row sort with flush scaling results show the most variation in throughput, this is because the row sort is most susceptible to large changes in sort time with random data. Depending on the distribution of the random records, one row could potentially take a long time to flush all of its records before it can start sorting the next temporary list.

3.5.2 Complete External Sort Results

In order to get a complete picture of these many-core sorts when used on an entire external database sort, the models described in Section 3.3 were used to compile data on the time and energy required to compute a 10-GB external sort. The phase 2 merge as well as other administrative tasks

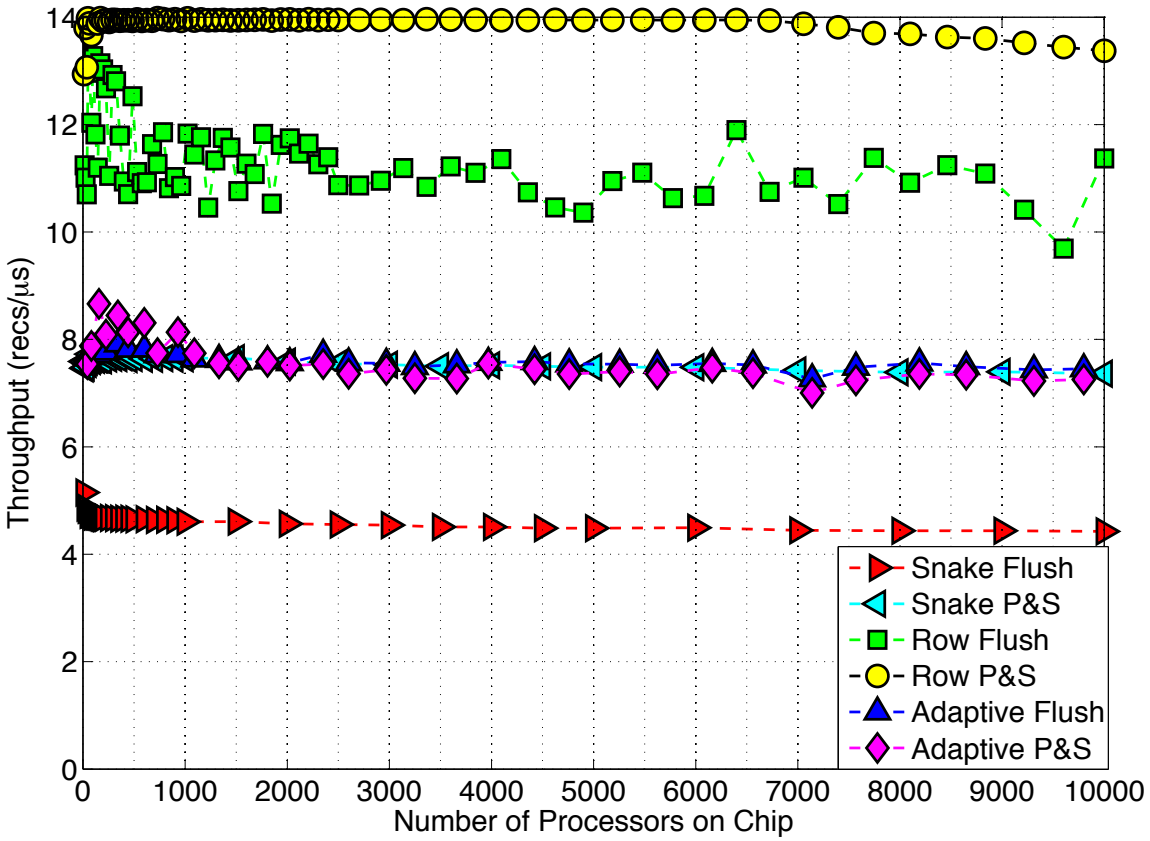


Figure 3.9: Throughput of the many-core array while scaling the number of processors with the phase 1 of the proposed sorts. The number of records sorted per temporary list is dependent on the number of processors, as described in Equations 3.1–3.3.

are performed on an Intel CPU, and phase 1 computations are performed on the given computational platform. The results can be found in Table 3.2. If multiple processing platforms were used, overlap can occur where parts of Phase 1 & 2 process concurrently if required data is ready. As no accurate energy or power numbers are given for Intel CPUs, 50% of the TDP for the Intel processors were used to determine the energy consumption [92].

The results of the full external sort models show the time to complete phase 1, the total time to complete the sort, the energy required by the processors to complete the sort, the total time multiplied by the area of the processing unit to highlight the tradeoff of smaller sized computational platforms, and the energy×delay, also known as energy-delay product (EDP), to highlight energy usage and execution time together.

Shown in Figs. 3.10–3.12, the phase 1 is a significant portion of the cost of an external sort performed on the Intel CPU and the Nvidia GPU. Figure 3.10 shows the energy required by the

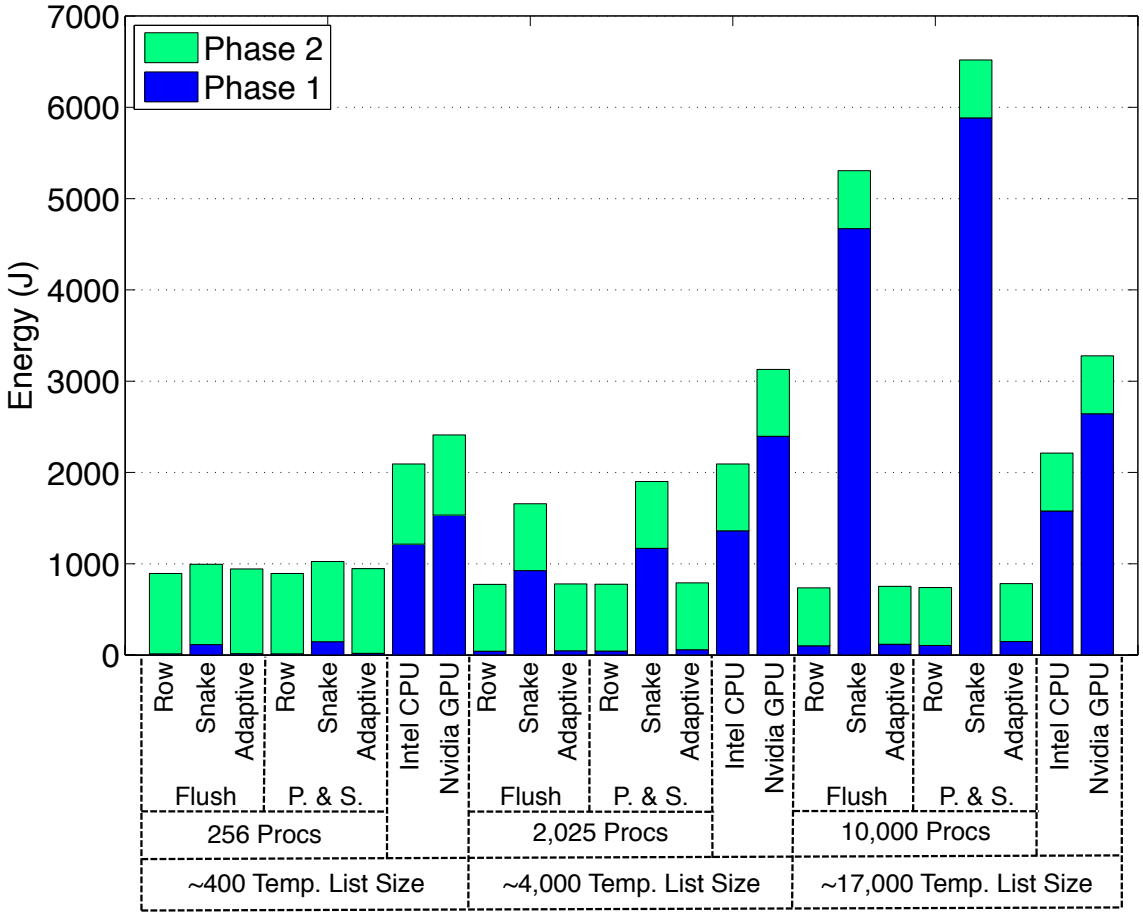


Figure 3.10: Total energy required by processors on a complete 10-GB sort.

processing unit to perform the different sorts. The energy cost of running the co-processor many-core chip is almost negligible in the graph when compared to the required power to operate the Intel processor for administrative tasks and phase 2 merge. This translates to extremely energy-efficient sorts on the many-core system, with the phase 1 row sort on a single 256-processor array taking over 83× less energy than the Intel CPU quick sort, and over 105× less energy than the Nvidia GPU radix sort. The phase one efficiency is overshadowed by the phase 2 energy, making the total energy cost for the most efficient many-core sort require more than 2.9× less energy than the comparable CPU sort and 4.4× less energy than the comparable GPU sort.

Shown in Figure 3.11, the many-core sorts are able to sort their 10 GB of data in less time than any of the comparison sorts. Table 3.2 shows the fastest row sort takes over 10× less time to perform phase 1 compared to the comparable quick sort on an Intel CPU and over 14× less than

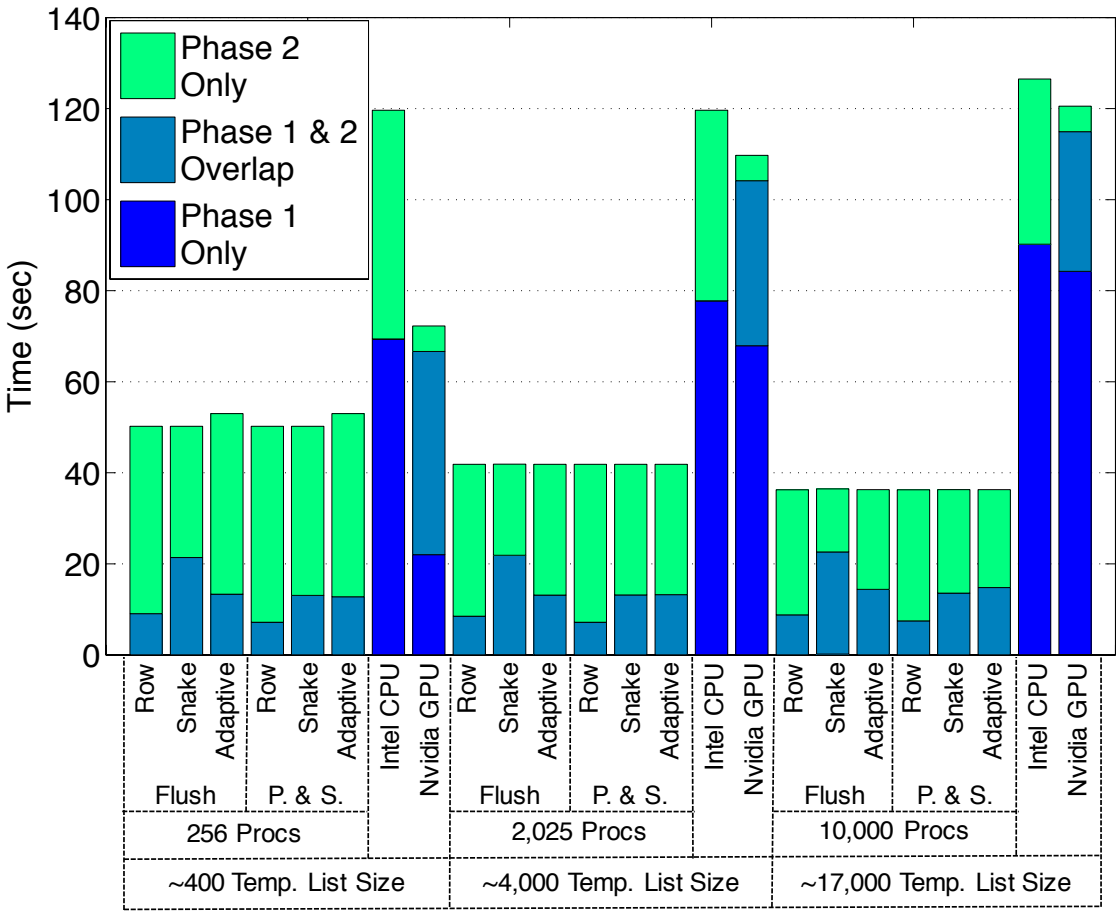


Figure 3.11: Time required to sort a complete 10-GB dataset. With two chips, there is some amount of time, marked Phase 1 & 2 in the legend, where both phase 1 and phase 2 are being computed at the same time on different chips.

the comparable radix sort on an Nvidia GPU. The largest difference in total time is found between the row sort with 10,000 processors showing over 3.4× smaller time that the comparable CPU sort, and over 3.3× smaller than the GPU sort. The sorting time is largely governed by the second phase merge with the proposed many-core sorts.

To highlight a benefit of using a small low powered many-core chip as a co-processor in a database system the total area×time metric was used, where the total time is multiplied by the total chip area of the processing unit. Figure 3.12 shows the tradeoff of scaling the number of processors. This analysis highlights that all of the many-core sorts are faster and more area-efficient than the CPU or GPU, except for the 10,000-processor many-core chip. The increased cost of area×time in phase 1 by increasing the number of processors to 10,000 is the penalty to decrease

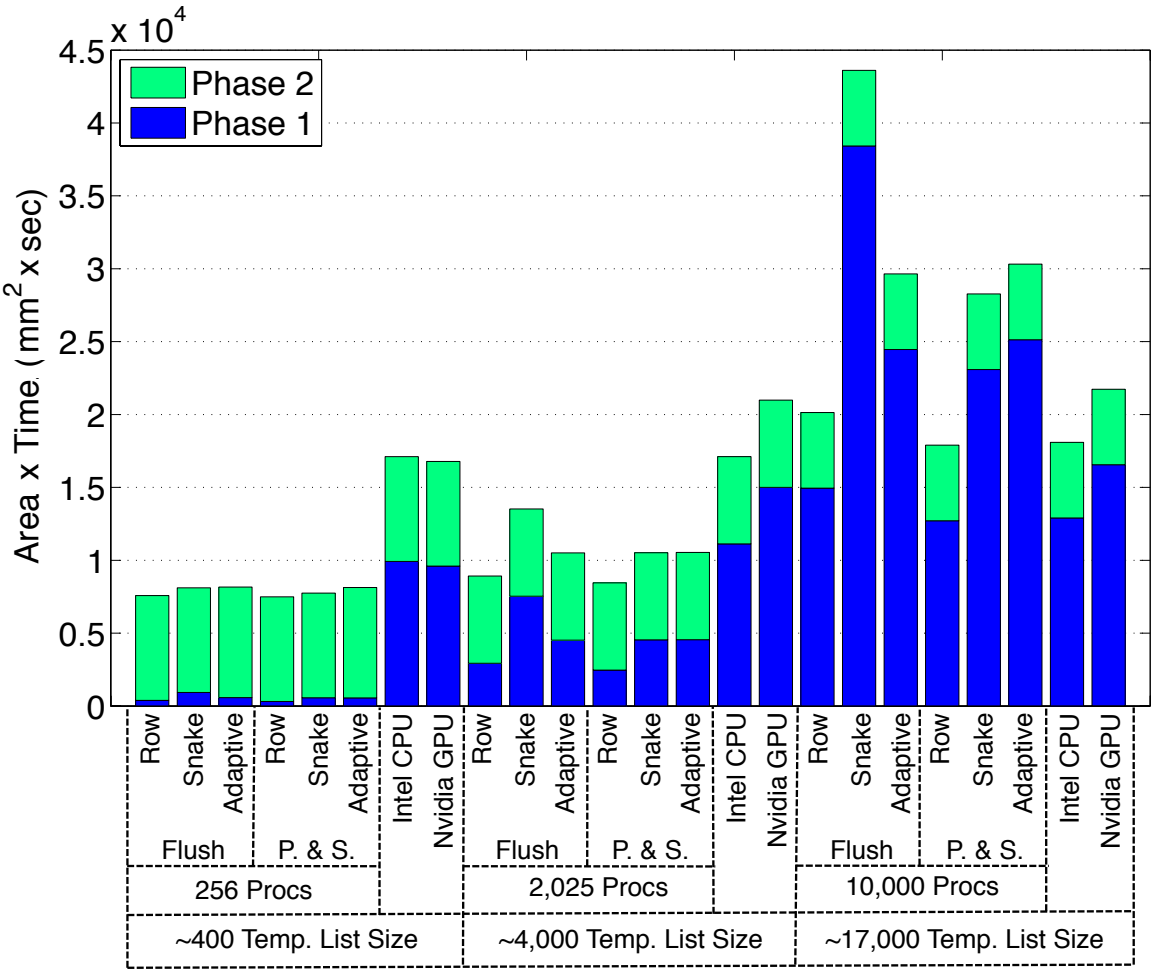


Figure 3.12: Area×time required by processors of a complete 10-GB sort.

the phase 2 sorting time, and decrease total energy, due to the longer sorted lists from phase 1. With a 256-processor many-core array, the row sort has over 2.2× less area×time than the Intel CPU quick sort, and over 2.2× less area×time than the GPU radix sort.

The energy×delay for the entire 10 GB sort is shown in Table 3.2, where the lowest energy×delay many-core sort was shown to require over 6.9× less energy×delay than the Intel CPU quick sort, and over 13× less energy×delay than the GPU radix sort.

As previously discussed in Section 3.5.1, the store and pass output protocol increases the throughput of all of the sorts. Figure 3.10 shows that with the phase 1 energy cost already at a small amount, the small increase in energy cost from the store and pass protocol has little to no effect on the total energy usage.

Comparison sorts were written and evaluated for the most direct comparison as they sorted

identical sized temporary lists and were built in the same technology node. However, the last entry in Table 3.2 displays the results from the 2007 JouleSort winner, CoolSort [32], in the Indy 10^8 record category, which was performed on a 65 nm Intel Core2 T7600. The same method used for the rest of the table was used to model the energy by multiplying 50% of the TDP by the sorting time [92]. No information on the phase 1 size, energy, or time were given for CoolSort. Because the JouleSort benchmark looks at system power, peripheral power and high I/O bandwidth is a large consideration, so fast processing was more important than low processing energy.

Table 3.2: Time, energy, area, and energy delay product for 65 nm platforms performing a 10-GB external sort of 100-B records.

Records Per Temp. List	Phase 1 Processor	Sort	Output Protocol	Phase 1 Time (s)	Phase 1 & 2 Time (s)	Phase 1 Energy (J)	Phase 1 & 2 Energy (J)	Phase 1 & 2 Area×Time (mm^2 s 10^3)	Phase 1 & 2 E×D (J s 10^4)
468	**256 Many-Core Processors**	Row	Flush	9.054	50.22	**14.5**	893	7.58	4.43
501		Snake		21.39	50.22	116	995	8.11	4.66
365		Adaptive		13.32	53.01	15.2	943	8.16	4.94
468		Row	Pass and Store	7.19	50.22	14.8	894	**7.49**	4.44
501		Snake		13.08	50.22	146	1030	7.75	4.61
365		Adaptive		12.79	53.01	19.1	947	8.14	4.94
468	Intel Core2 T7700	Quick		69.41	119.6	1210	2090	17.1	12.8
468	Nvidia 9600M GT	Radix		66.67	72.25	1530	2410	16.8	14.6
3919	**2,025 Many-Core Processors**	Row	Flush	8.516	41.85	42.9	775	8.92	3.10
4001		Snake		21.89	41.89	926	1660	13.5	5.09
3657		Adaptive		13.14	41.85	46.4	779	10.5	3.13
3919		Row	Pass and Store	**7.163**	41.85	44.3	777	8.45	3.10
4001		Snake		13.18	41.85	1170	1900	10.5	4.61
3657		Adaptive		13.23	41.85	58.4	791	10.5	3.14
3919	Intel Core2 T7700	Quick		77.80	119.7	1360	2090	17.1	13.7
3919	Nvidia 9600M GT	Radix		104.2	109.7	2400	3130	21.0	28.0
19 704	**10,000 Many-Core Processors**	Row	Flush	8.796	36.28	102	**737**	20.1	2.39
20 001		Snake		22.60	36.46	4670	5310	43.6	12.9
16 737		Adaptive		14.39	36.29	118	753	29.6	2.47
19 704		Row	Pass and Store	7.478	**36.27**	105	739	17.9	**2.38**
20 001		Snake		13.59	36.29	5880	6520	28.3	10.3
16 737		Adaptive		14.78	36.30	149	783	30.3	2.52
19 704	Intel Core2 T7700	Quick		90.2	126.5	1580	2210	18.1	16.5
19 704	Nvidia 9600M GT	Radix		114.9	120.5	2640	3280	21.7	32.7
—	Intel Core2 T7600	CoolSort [32]		—	86.6	—	1470	12.4	12.7

Chapter 4

Physical Design of GALS Processor Arrays in the 1000-Processor Era

4.1 Design Goals

When designing our programmable processing system, six key goals were set: i) processors that occupy very small chip area, ii) processors that dissipate very low energy when active which is relatively easy when they are small area, iii) processors that dissipate very low energy when idle, iv) no algorithm-specific hardware in the programmable processors; v) GALS clocking with a fully-independent digitally-programmable clock oscillator inside each processor tile with no phase-locked loops (PLLs), delay-locked loops (DLLs), or crystal oscillators and the ability to change frequency (of course constrained below the maximum operating frequency, f_{max}), halt, and restart the clock arbitrarily; and vi) low energy, high rate, inter-processor communication.

These features resulted in several key outcomes: i) massive numbers of processors per chip allowing processors to be used for tasks never envisioned with traditional architectures; ii) performance and efficiencies that excel across a variety of application domains, and iii) an architecture that has some advantages in dealing with features of current and future fabrication technologies such as adapting to process, supply voltage, and temperature variations, operating with fabrication defects, and potential for self-healing after wear-out failures.

4.1.1 Design Methodology

Early design space exploration was carried out in a many-core simulator written in C++. The register-transfer level (RTL) for KiloCore was written in the Verilog hardware description language (HDL) and simulated using Cadence NC-Verilog. The physical design of the chip was done in a bottom-up fashion. The lowest level blocks were synthesized using Synopsys Design Compiler to provide a gate level netlist, which included instantiations of ARM standard cells for IBMs 32 nm partially depleted silicon on insulator (PD-SOI) process. The gate level netlist was placed and routed using Cadence Encounter. The lower-level blocks were then instantiated and placed into an array, where inter-processor and inter-memory communication was routed in Encounter. Finally this array was placed into the total chip design, which includes I/O drivers, electrostatic discharge (ESD) clamps, and large capacitors attached to the power rails to reduce supply noise. Final components of the chip were added in Cadence Virtuoso. The design was tested and validated using Mentor Graphics Calibre and Cadence UltraSim.

4.2 Fine-Grained Many-Core Power Distribution

One of the key design elements of our many-core processor array design was the use of homogeneous tiles. A single design was made, and it was repeated as many times as was desired, or would fit in the given die area. Because of this, the power grid and distribution was considered from the outset of the design, as a robust power delivery system was desired. Therefore it was dedicated the top two layers of our metal stack to global power distribution, as those two layers are the thickest, and can be made thicker than lower level metals. Therefore these wires were the lowest resistance, to minimize voltage drop over the power grid, and the highest capacitance value, to help smooth out spikes in current draw across the grid. The amount of wire that could be dedicated to the power grid was limited both by metal density design rules and the location of the controlled collapse chip connection (C4) bumps. The regular pitch of the C4 bumps further required the global power grid to adopt a similar pitch, which did not line up with the size of a single processor tile.

Each of these small homogeneous processor tiles have the potential for a high current draw. The very conservative value of 500 mA per core was considered during the power design stage. It was determined that simply making a few connections where the local power grid crossed the global

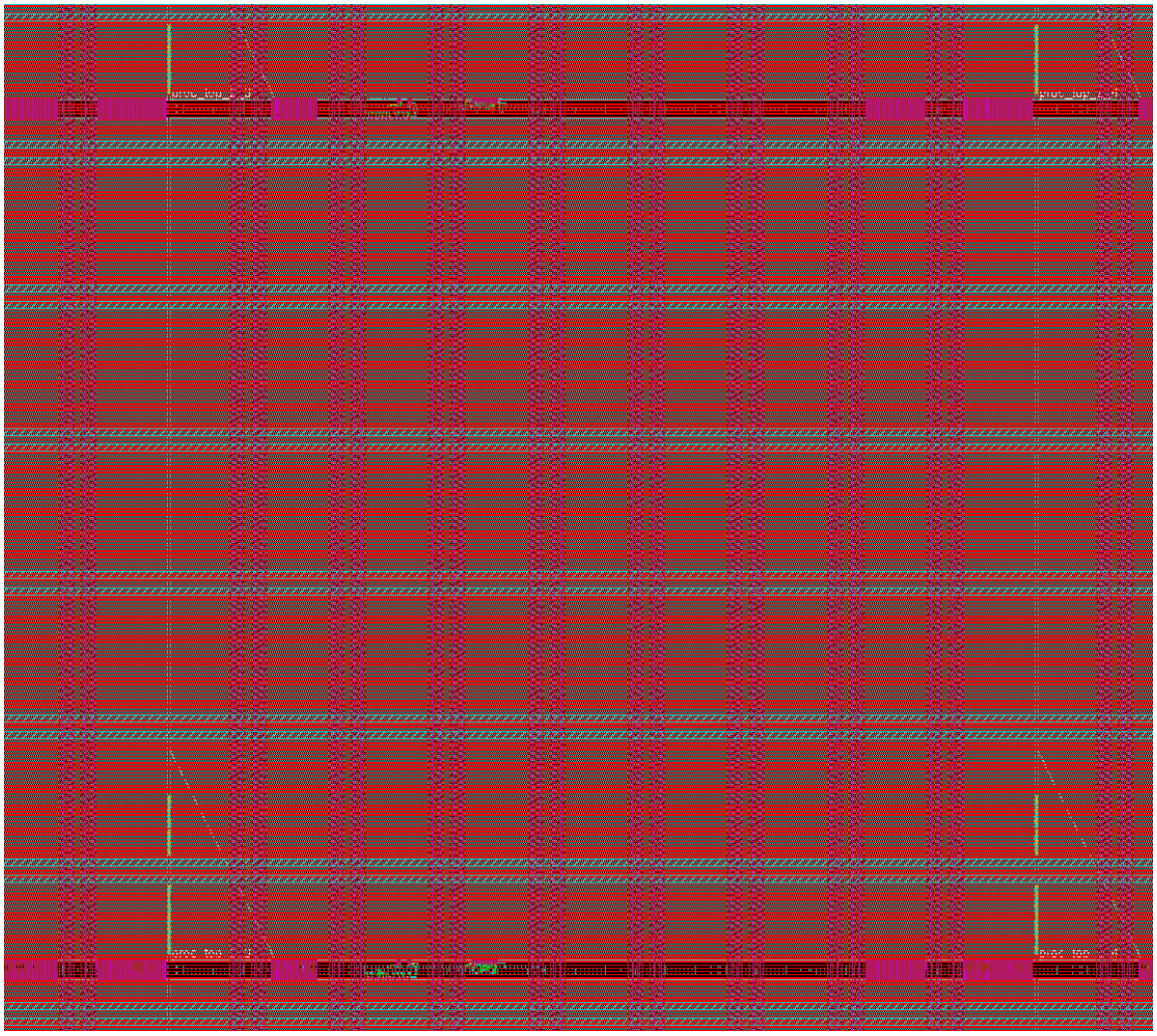


Figure 4.1: Plot from Encounter focused on single processor, showing the local power grid and how it extends through its neighboring processors.

power grid was insufficient. Therefore the local power grid was designed in such a way that the lower level metals that make up the local power grid, layers 4–9, could be extended in all 4 directions, and connect through all neighboring processors, thereby creating a robust local power grid, even if a processor module didn't line up well with the global power grid. The local power grid for KiloCore can be seen in Figures 4.1 and 4.2. Notice how the local power grid extends through neighboring processors, as well as the lowest level metal layer power rails for the individual standard cells. The global power grid for KiloCore can be seen in Figure 4.3, where the top level metal layer is used both for the global power grid and chip I/O signal wires. In the design of KiloCore, the power grid design was slightly complicated by the C4 locations on the package that was to be used with the

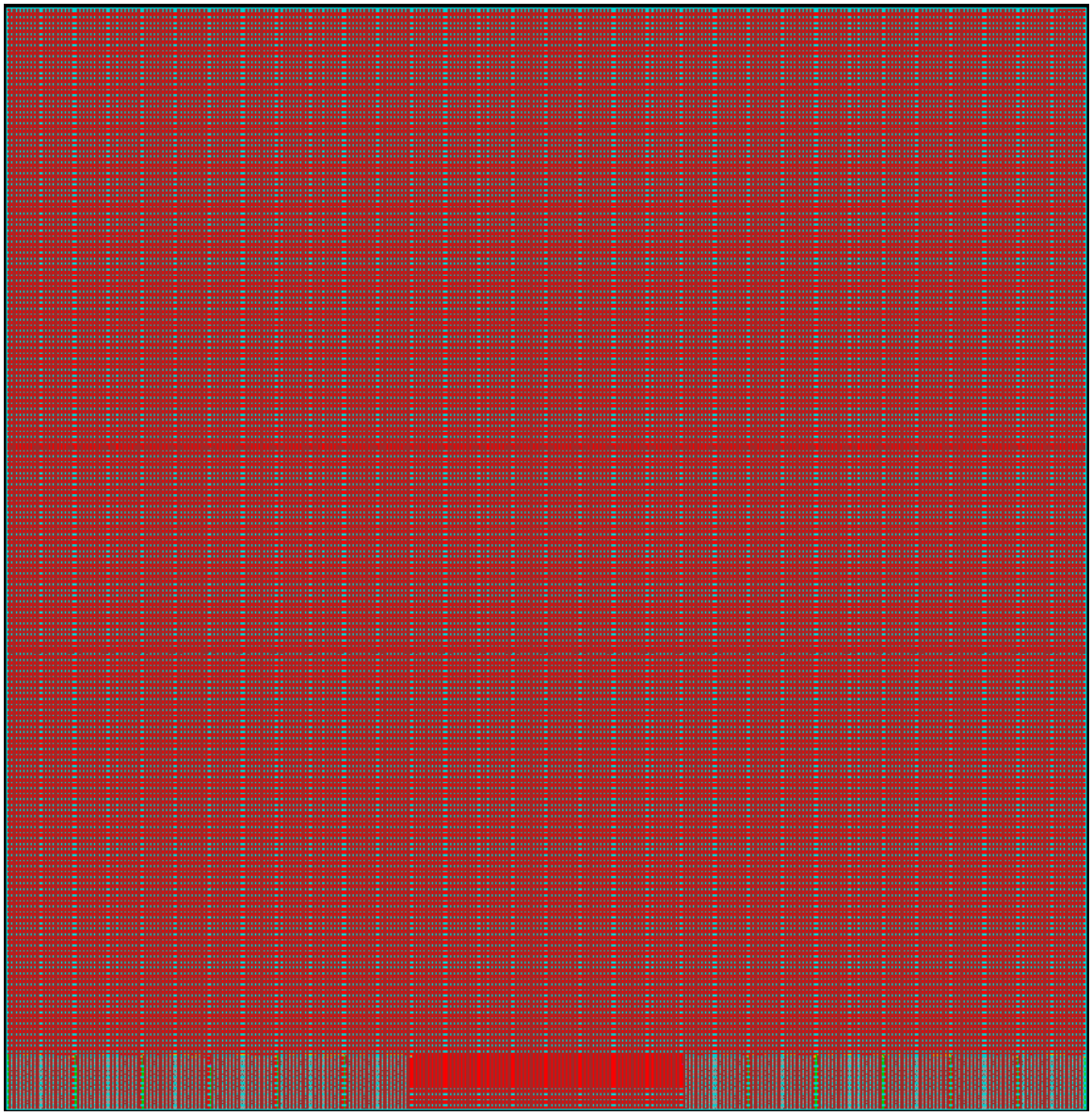


Figure 4.2: Plot from Encounter showing the entire KiloCore array. The local power grid is displayed and how it extends through its neighboring processors and independent memories. The global power grid is not displayed here, but can be seen on Figure 4.3.

chip. As shown in Figure 4.4, the signal I/O C4 bumps were in the middle of the chip, so long wires were required to be routed from the I/O drivers on the periphery of the chip, to these C4 bumps, which blocked the power grid from those locations, as they had to share the same top two metal wires. KiloCore2 was going to use a custom designed package, this was not an issue, as the area was filled with C4 bumps, so the signal bumps were much closer to the I/O drivers.

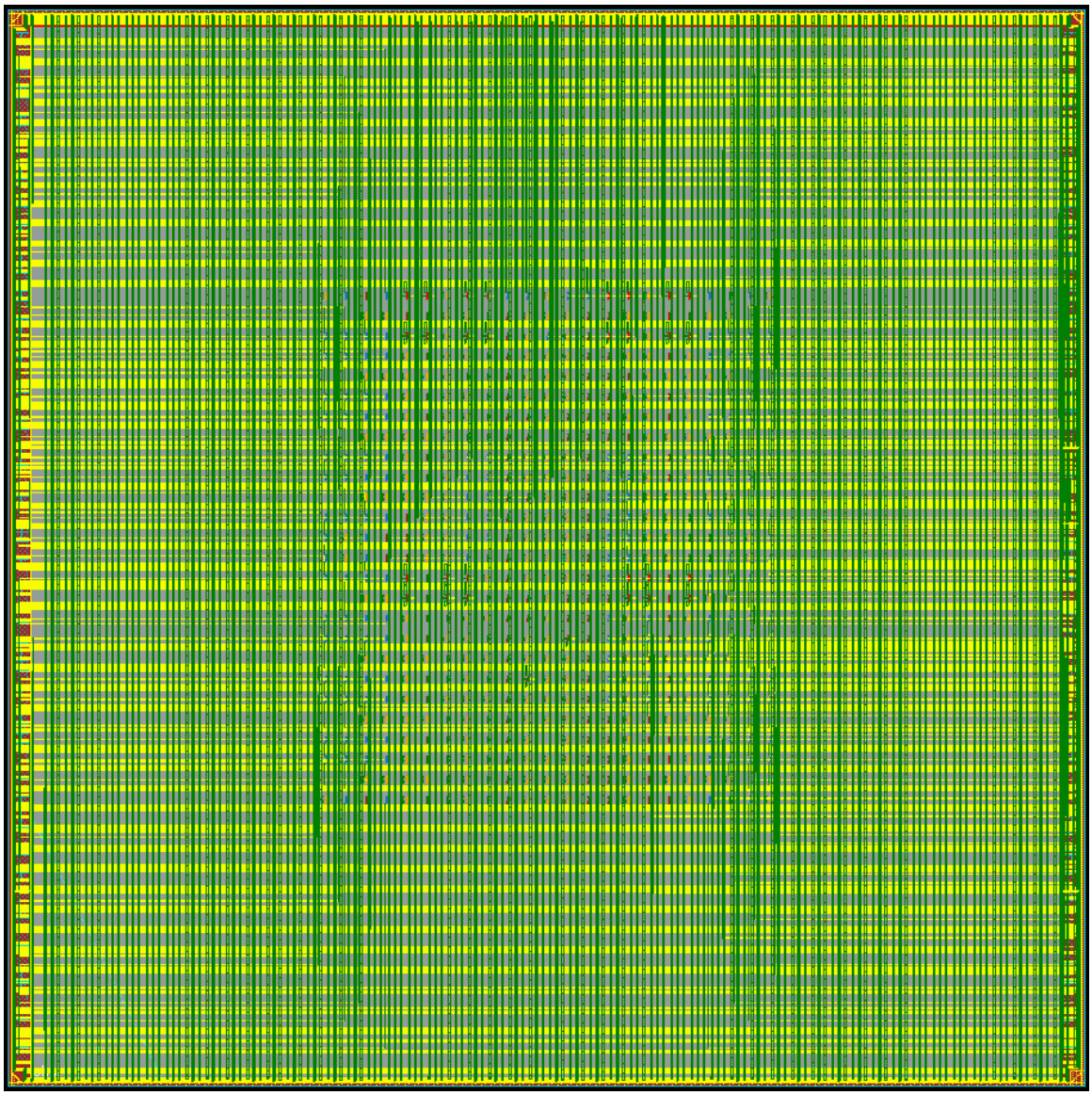


Figure 4.3: Plot from Encounter showing the entire KiloCore chip. The top two layers of metal were used both for the global power grid, and for the routing of global signals to the C4 bumps for chip I/O.

4.2.1 Dynamic Voltage and Frequency Scaling Considerations

KiloCore2 contains three separate power supply rails, high, medium, and low. Each processor can dynamically switch between any of these three power rails independently, which the high supply rail also supplies all circuitry that is always on. The high supply rail was chosen for always on so that communication signals will always be greater than or equal to the highest voltage inside of a processor core. This brings up a number of issues that must be considered.

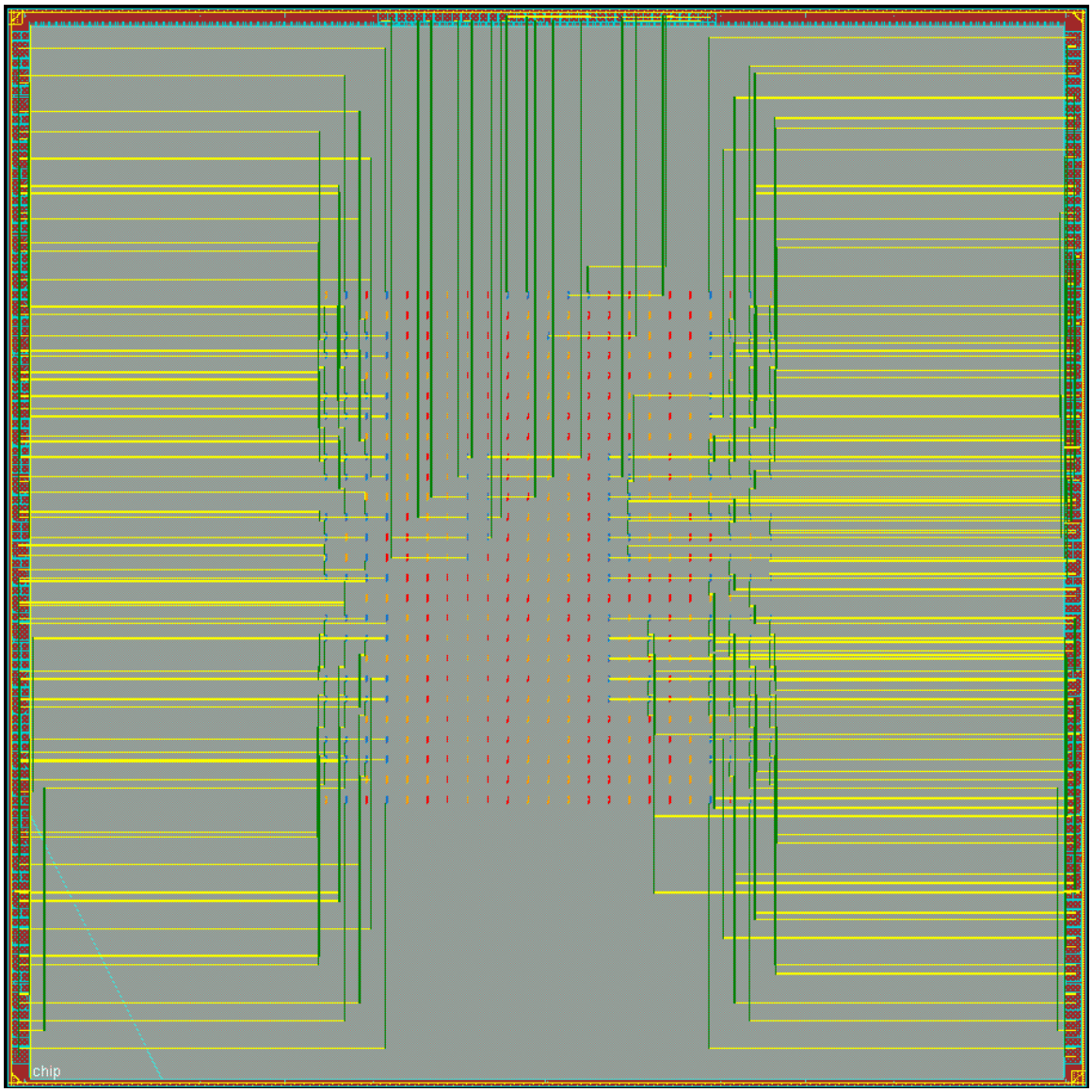


Figure 4.4: Plot from Encounter showing the global signal wires of KiloCore as they are routed from their peripheral I/O drivers to their matching C4 bumps for chip I/O.

With 5 power rails (high, medium, low, I/O, and ground), the layout of the power grid must be carefully considered to allow for enough metal to be devoted to each rail, so as to allow enough current to be delivered from the C4 bumps to where it needs to go. One common ground is used, therefore this rail will have more of the power grid dedicated to it, than the other rails. It is not expected, by design, that the four Vdd rails will all supply maximum current simultaneously, so it is not necessary for the common ground to be able to transmit the maximum current of all of the other rails combined. Both because the high rail needs to supply the always on circuitry,

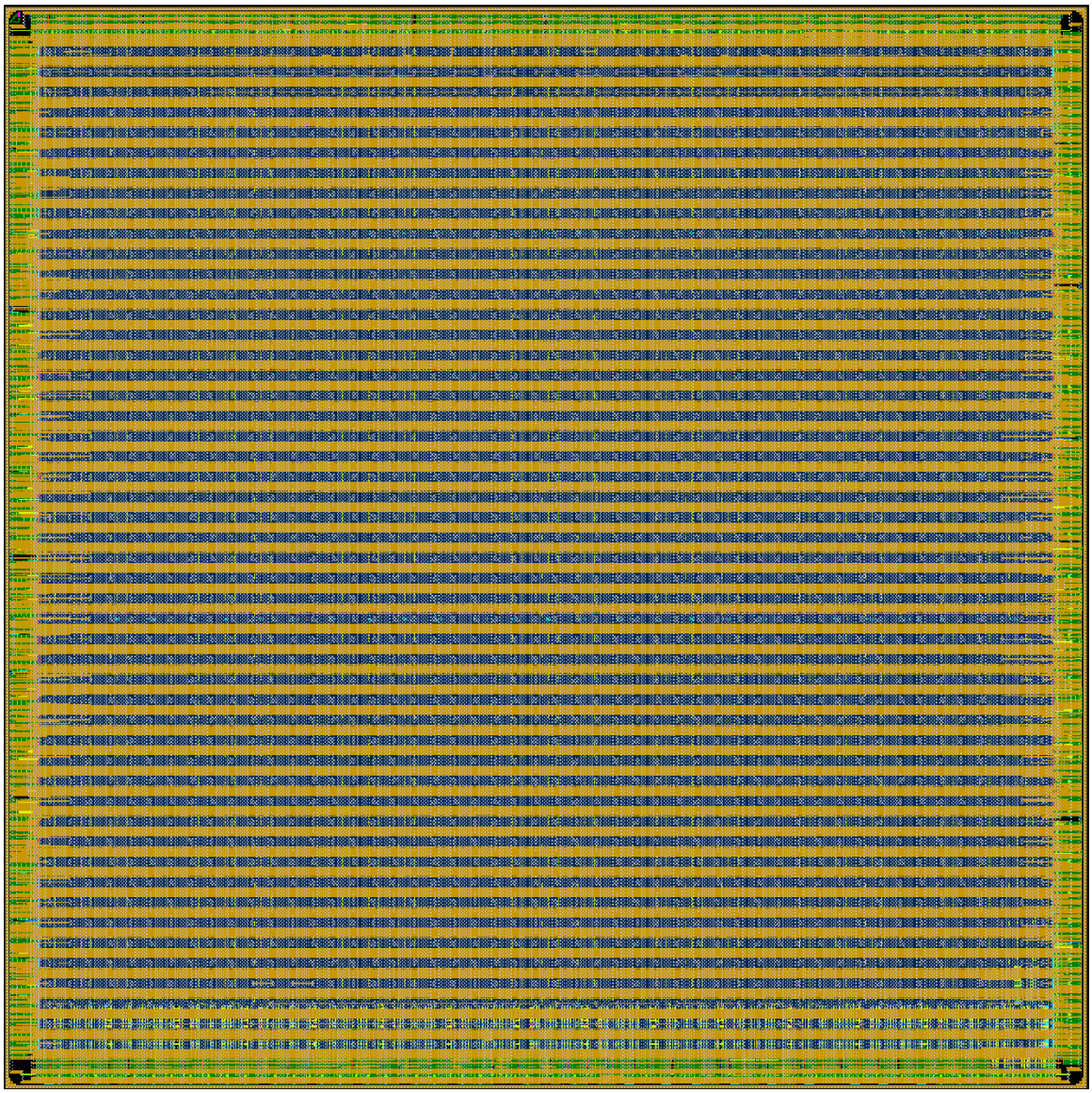


Figure 4.5: Plot from GDS showing the entire KiloCore2 chip. The top two layers of metal were used both for the global power grid, and for the routing of global signals to the C4 bumps for chip I/O.

and because it is expected to have the highest voltage by design, the second most metal must be dedicated to the high rail. A plot of the entire KiloCore2 die area from Encounter is shown in Figure 4.5.

To enable the core voltage to be switched between the three power rails, power gate headers were applied to connect the power rails to the local core power grid. As power gates are transistors that physically reside in the substrate, this means that power from the power grid, which, as discussed in Section 4.2, is desirably a high level metal, must traverse many levels of resistive contact

vias to supply current to the gate. This, as well as simply traversing the gate, creates an undesirable voltage drop that must be minimized. There are two common methods of implementing power gates, either in a ring around the outside of a design to be power gated, or dispersed throughout a processor. For KiloCore2, it was decided that the second method was preferable, both because the power gates given were small standard cell sized, and for the benefit of minimizing the need for current to travel down the metal stack to the power gate, and back up the metal stack to a local core power grid.

The minimum number of power gates and the minimum dispersal distance was determined for a worst case overall voltage droop of 5%. This was determined both with hand calculations and computer simulations, assuming a very conservative 500 mA current. The resulting power gate placement is shown in Figure 4.6. Figures 4.7, 4.8, and 4.9 show an example of current flow from the high power rail, down to the power gate, to the logic gates, and finally up to the common ground.

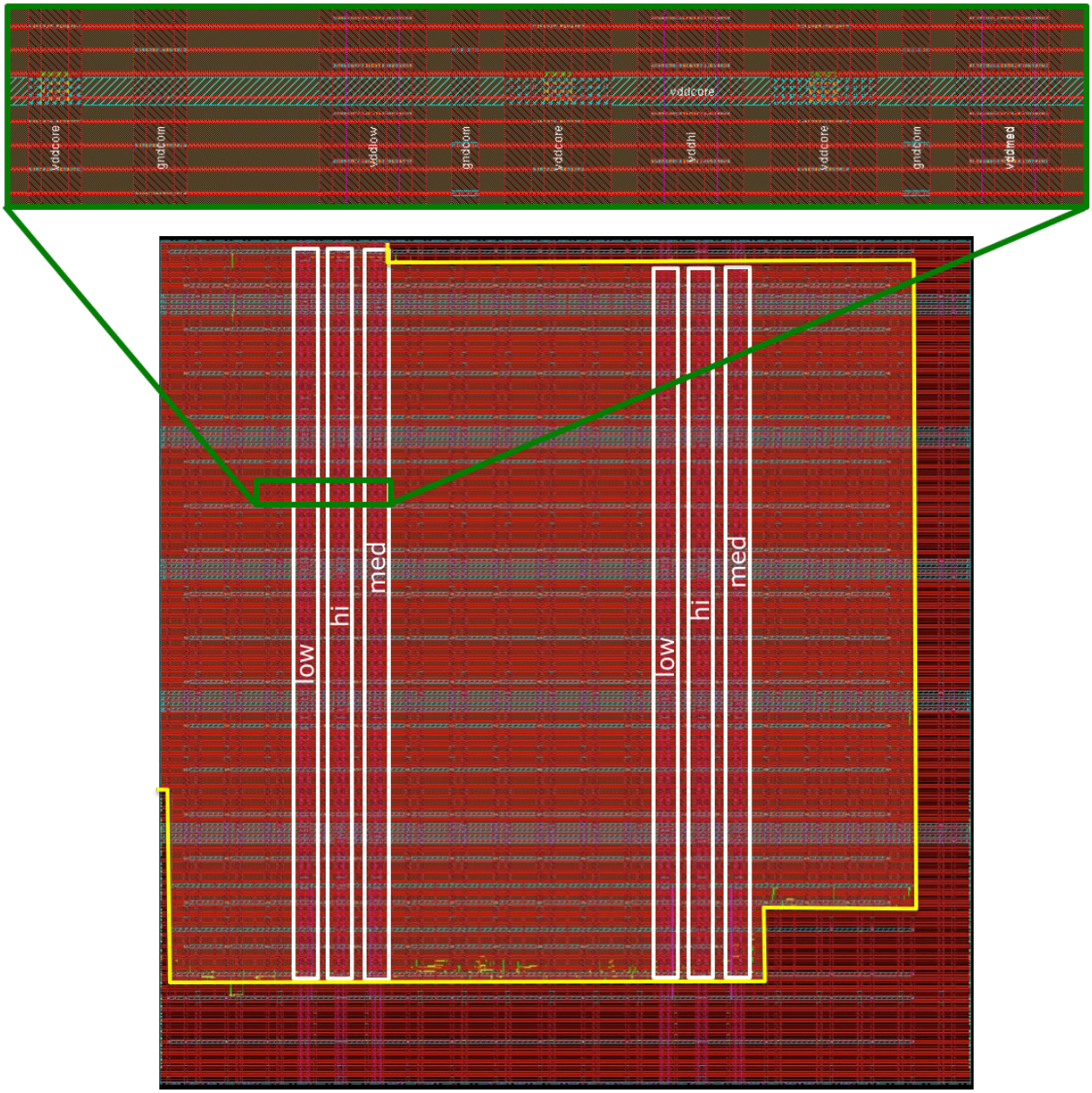


Figure 4.6: Plot from Encounter showing the entire KiloCore2's local power gate placement, and zooming into one of the rails to take a closer look at the power gate operation in Figures 4.7, 4.8, and 4.9. The power gate columns are highlighted in white, and the local core power domain is highlighted in yellow.

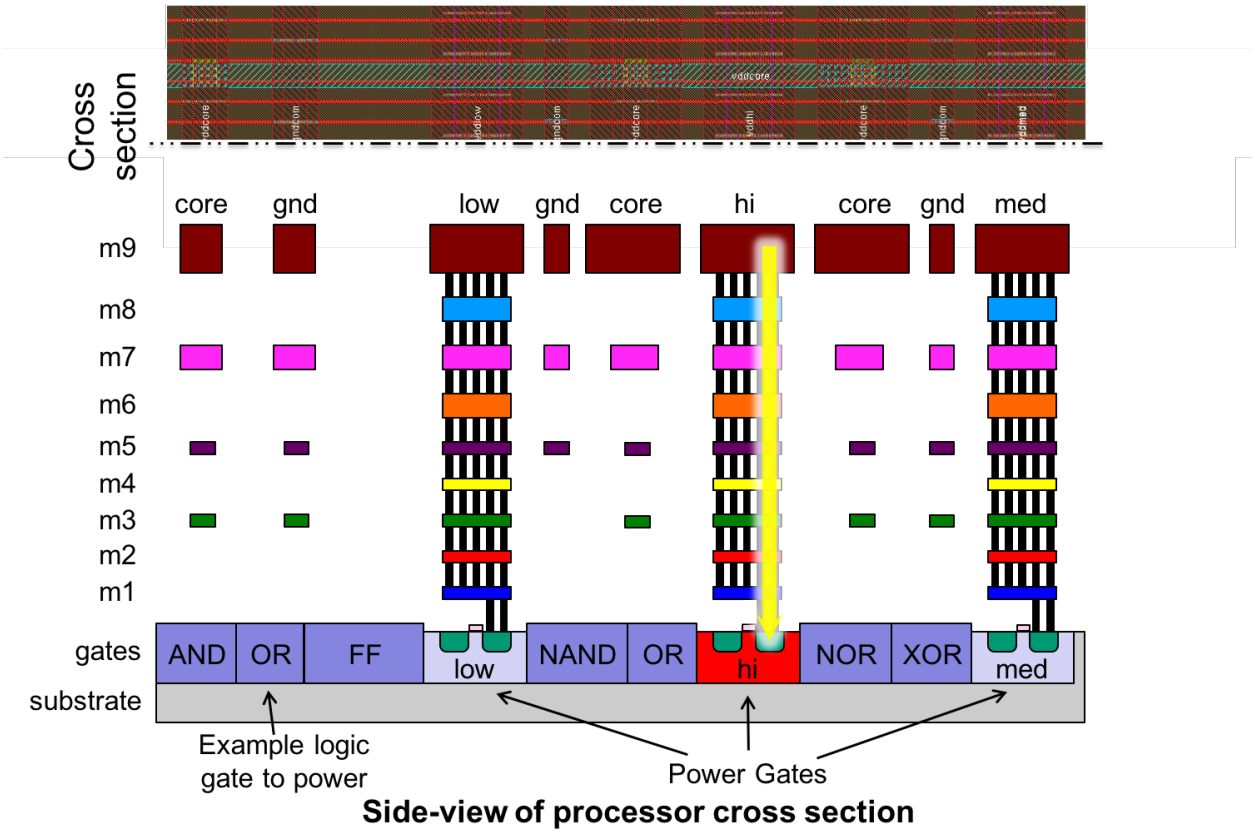


Figure 4.7: Illustration of current going from the hi power rail down to a power gate.

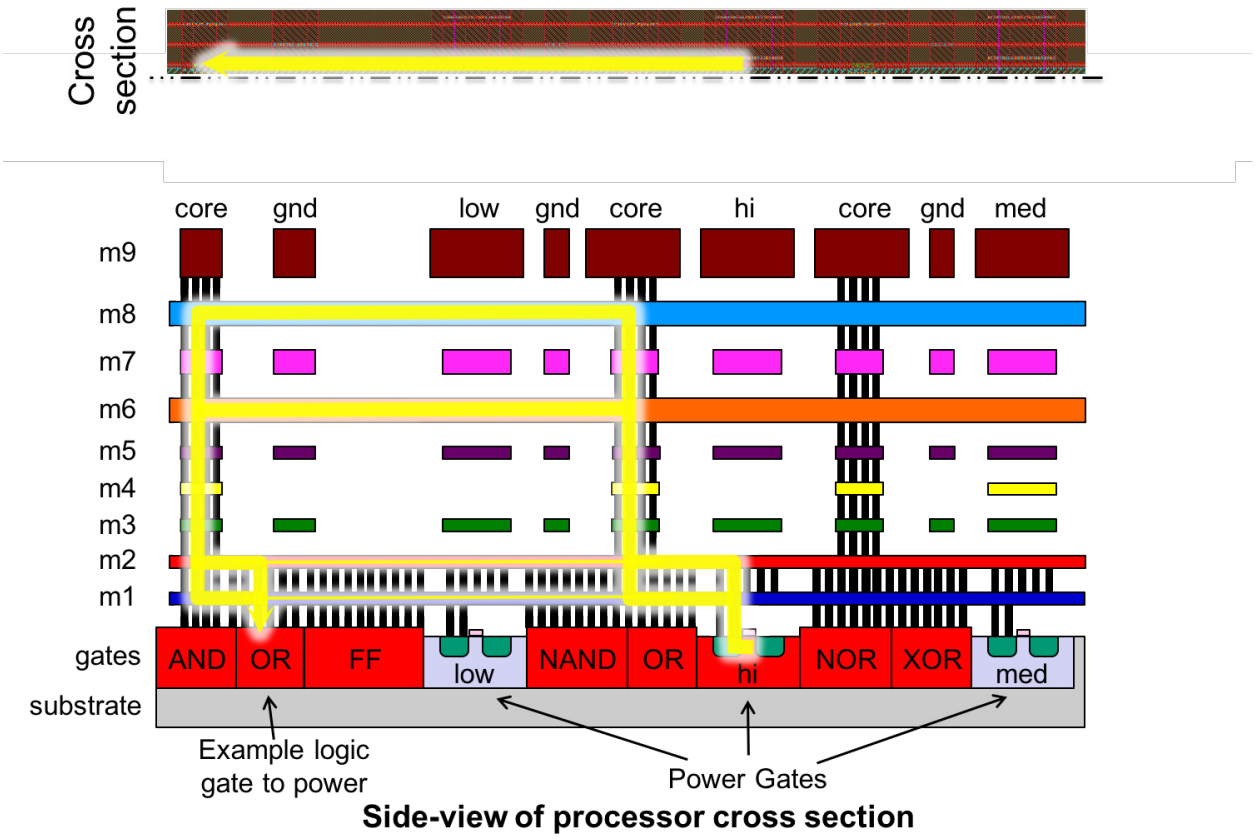


Figure 4.8: Illustration of current going from a hi power gate to example logic. Note the cross section has moved from Figure 4.7. Notice the path of least resistance is to the power gates is through the local core power grid, but power can still be delivered through the logic gate's power rails.

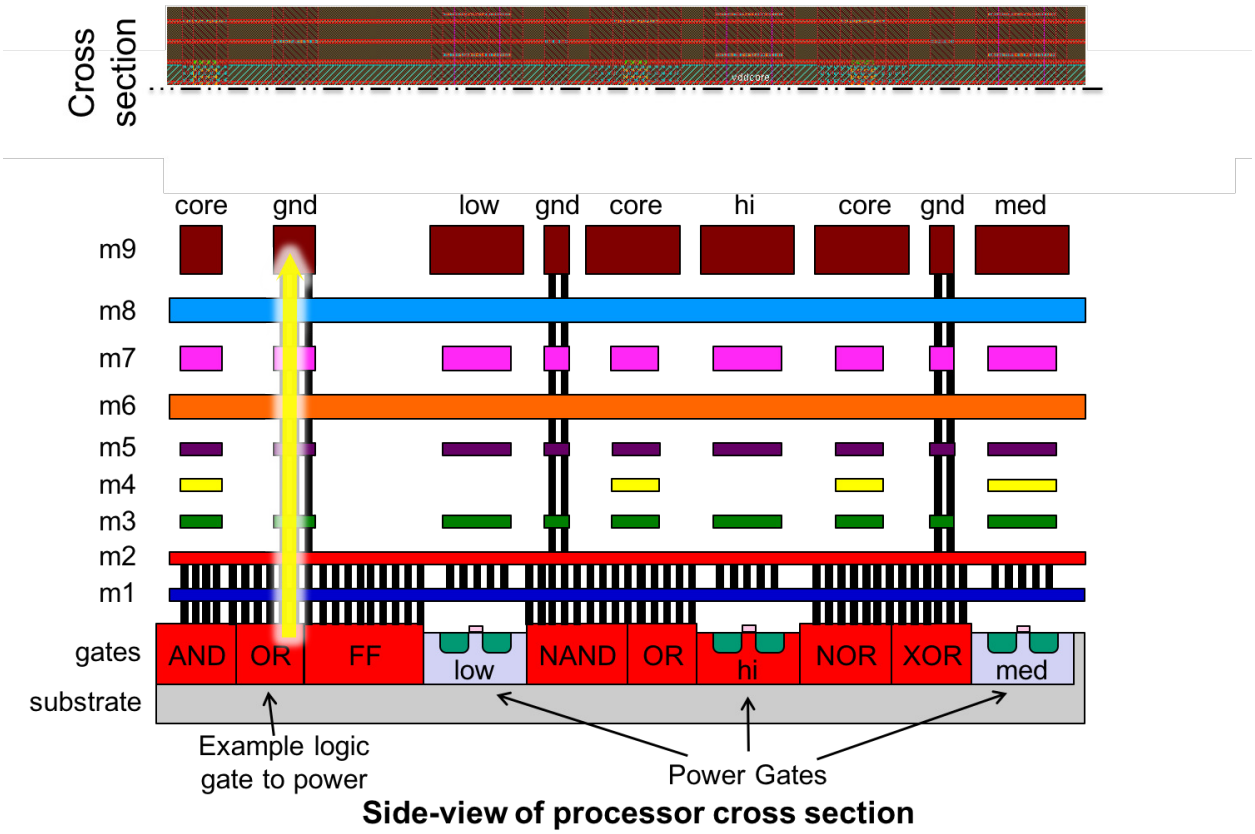


Figure 4.9: Illustration of current going from example logic to the global power rail for the common ground. Note the cross section has moved from Figure 4.8.

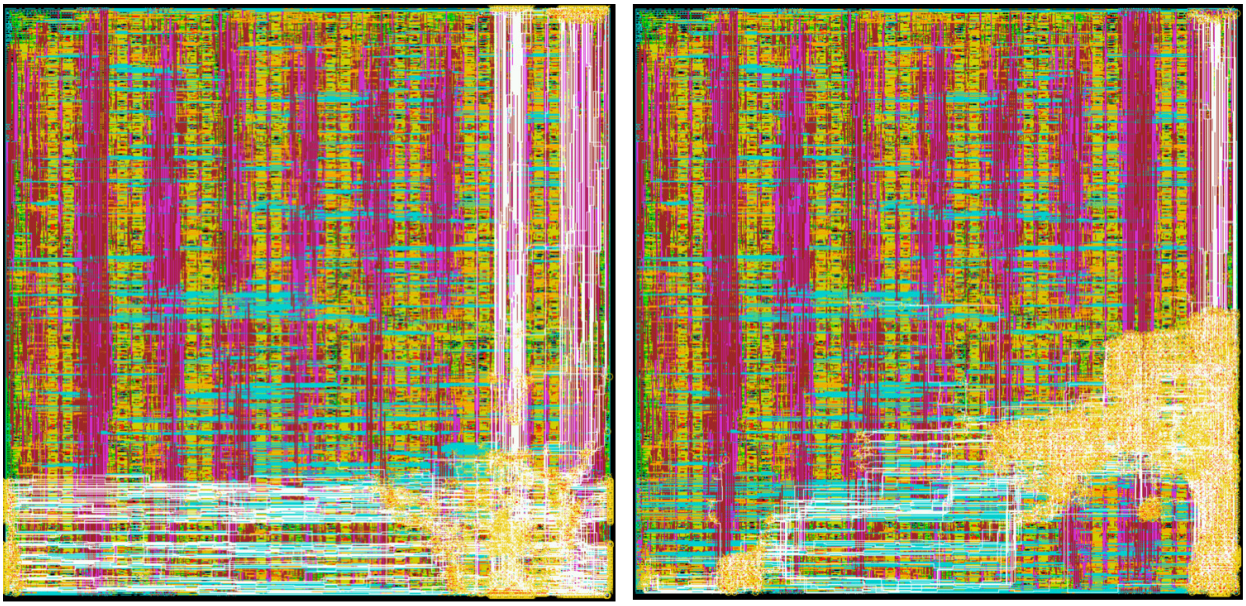


Figure 4.10: Plot of the layout of KiloCore with the communication paths highlighted.

4.3 Many-Core Placement and Routing

4.3.1 Internal Module Considerations

In both chip designs, routing of communication signals, both for the circuit switched and router networks, were encouraged to go along the bottom and right side of the processor tiles. This was to keep the long wires from congesting the routing in the middle of the processor, as well as keep these high activity factor wires far from the oscillator, which was placed in the top left, so as to minimize noise in the clock. This was accomplished by defining the local processor I/O ports to the left bottom, top right, and right bottom sides, and by creating routing properties to prefer wider, high level metals with minimal jog for the communication wires. Figure 4.10 shows a plot of the layout of KiloCore with all of the wires corresponding to the processor communication ports.

For KiloCore2, with its switchable core local power domain, there are parts of the processor that must always be on, such as the DVFS controller to manage the voltage switching, the router to

Figure 4.11: Plot of the layout of KiloCore2 showing the power and ground wires, and the local core power domain which is highlighted in tan. Note the small power wires near the power domain boundaries, which are used to power the level shifter cells that transfer communication wires from the always-on power domain, high, to the local core power domain, which could be high, medium, or low.

allow for communication to continue when a processor is off, as well as the circuitry used for the circuit switched network. The ideal location for the router is at the intersection of the communication

routing paths, which is the bottom right corner of the processor. This hypothesis was confirmed by running place and route with no spatial requirement on the router module. The ideal shape for a digital integrated circuit module is a square, to minimize wiring distances, therefore a square was required in the always-on power domain. Enough space was required to fit the above mentioned circuitry that was in the always-on power domain, so this domain was given the area mainly on the bottom of the tile, with some space devoted to it on the right side of the design. With hundreds of inter processor communication ports, the processor I/O port locations traversed a large portion of all four sides of the processor. These communication signals are crossing a long distance, and as such, need to be buffered, therefore a small amount of the always-on power domain was apportioned to cover a small area around the processor tile input and output ports. The power rails are shown in Figure 4.11 along with the tan shape identifying the local core power domain.

4.3.2 Chip-Scale Design Considerations

With a thousand instantiations of a single homogeneous processor design in a single die, small wasted space adds up quickly. Therefore the physical design of a single core is carefully considered, and optimized, with numerous place and route computational runs with slight tweaks to the processor dimensions, until a size was found that was as small as possible, while still providing an optimal critical path, and an amount of wire congestion that did not adversely affect performance. Similarly, the space in between processors needed to be minimized, as the KiloCore design contained 31 rows and 32 comlumns, this space again adds up quickly. Therefore the space was minimized as much as possible, leaving only 7 standard cell heights between each row, which is used both for tie high and tie low cells for the processor number to be input into the processor, as well as for buffers for signals. The horizontal space between the columns is only 0.4 µm, which allowed for the standard cell level power rails to be continued in between processors horizontally, as seen in Figure 4.12. As mentioned in the previous subsection, inter-processor I/O pins locations were carefully considered, including global signals, such as programming ports, external clock signals, and test signals. The inter-processor I/O signals can be seen in Figure 4.12.

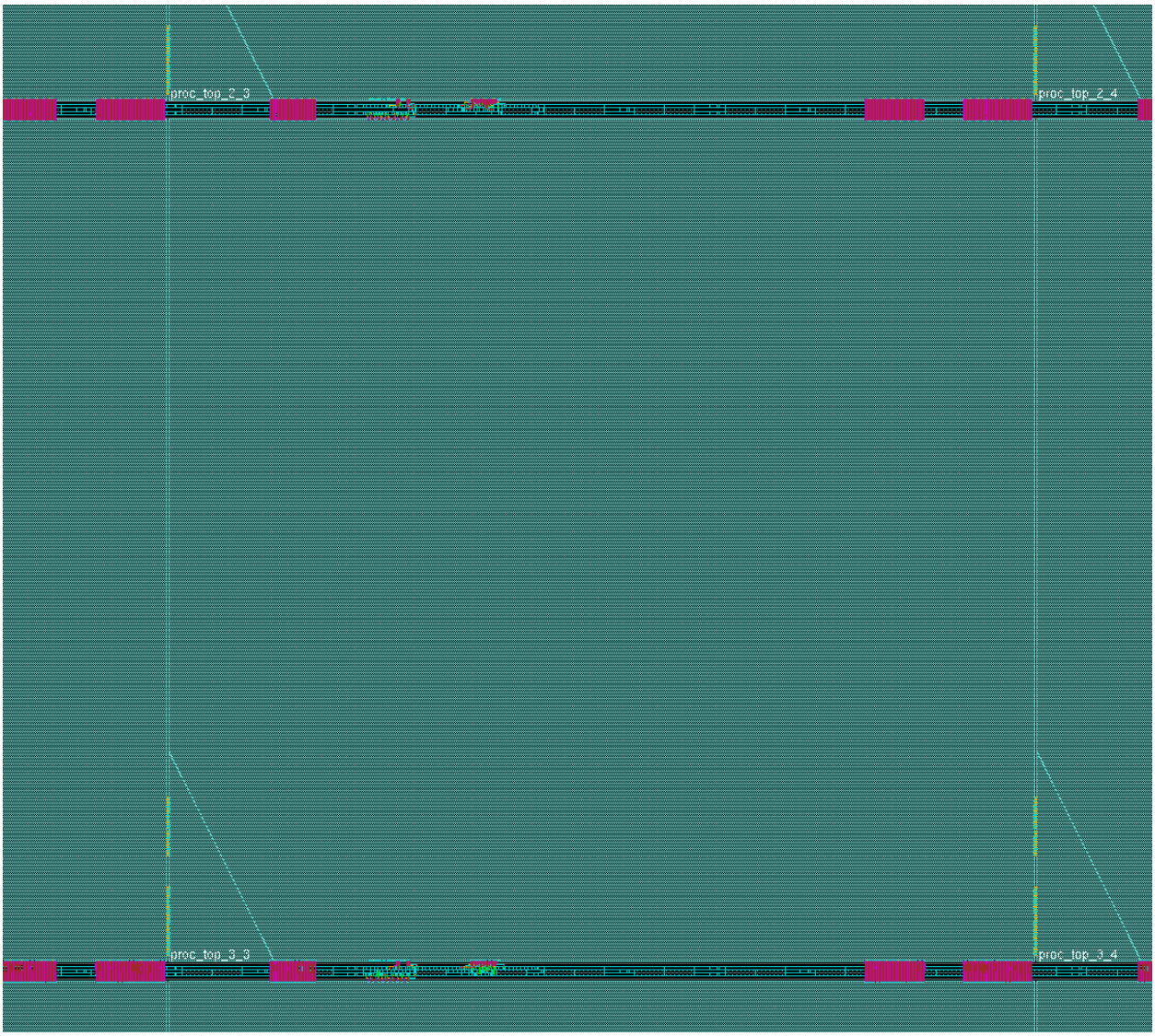


Figure 4.12: Plot of the layout of KiloCore showing the abutment of cores in array. Note the vertical space between processor tiles is 7 standard cell heights which allows for for buffers and tie high/low cells to provide the processor number. The horizontal space between tiles is just 0.4 µm, with standard cell level power rails continued horizontally between processors.

4.4 Inter-Processor Communication

4.4.1 Many-Core GALS Communication

The processor array connects processors via a two-dimensional mesh grid, a topology which maps well to planar integrated circuits and scales simply as the number of processors per die increases Figure 4.13. Processors communicate with each other via a pair of statically-configured circuit-

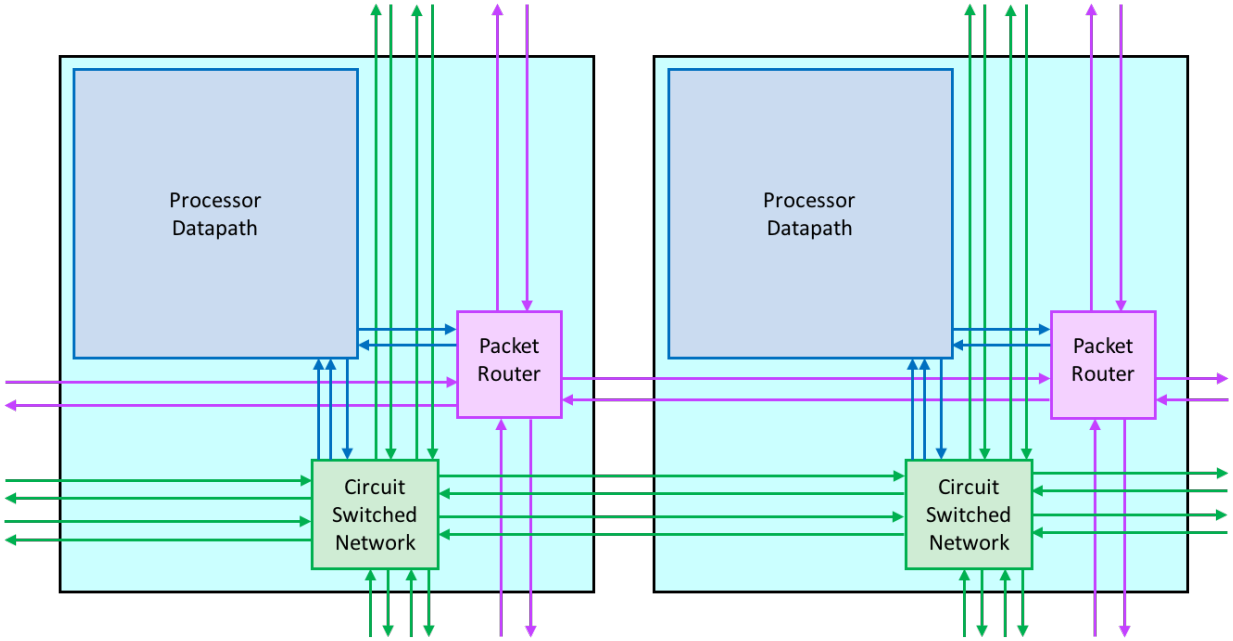


Figure 4.13: KiloCore's communication paths. There are two circuit switched links in each direction from each processor tile. Each link contains a 16-bit dataword, a valid signal, a clock, and a return request signal. Two inputs can be connected to the two dual-clock FIFOs inside the internal datapath of the processor. Inside the circuit switched network, signals can be routed in any direction either directly, or after being registered. Using either the packet router or the circuit switched network, data can be transmitted across processor tiles, without ever entering the clock domain of a local processor.

switched networks and a dynamically routed packet network. The circuit network supports direct links between distance processors, with optionally inserted registers to maintain data integrity over long distances. Links utilize source-synchronous signaling, in which a reference clock is transmitted with data. Each circuit or packet link terminates in a dual-clock FIFO memory structure which safely 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.

4.4.2 Timing Considerations

Current CAD tools are not designed to handle timing between 1000 processor tiles, where a communication timing path exists from any of the 1000 cores, to any other of the 999 cores. The timing computations of the CAD tools could not handle the computations, and simply

stalled indefinitely. Therefore timing was disabled on inter-processor communication, and timing requirements, which were thought to be robust, were placed on the timing of a single processor's design. Soon after the tapeout of KiloCore, on closer evaluation, it was determined that there existed some hold time violations between a number of the communication paths in the inter-processor communication networks. These timing violations luckily were not completely chip disabling, as this only stopped communication if the hold time violation occurred on the communication's valid signal. As long as the valid signal had no timing issues, the communication could simply be sent twice, with the receiving processor using the second of every received dataword. In the case the the valid signal had a timing violation such that it was indeterminate which clock period it would arrive in, the particular communication path was unusable. This was not a large percentage of paths, and if it was the case, the mapping tool can simply take this into consideration, and not use these certain ports.

This led to the careful consideration of inter-processor timing for the KiloCore2 design. A robust method was integrated into the flow, as mentioned in Section 4.5. This method included creating a smaller test chip, of 9 processors, to allow the CAD tools to adequately test the communication timing. A 9 processor chip was laid out, and the design requirements of the individual processor tile was modified until the CAD tool no longer needed to add any buffers to modify timing in between processor tiles. In this way it was determined that the timing inside the boundaries of a single tile was sufficient, and was then used to layout a full sized array.

4.4.3 Dual Clock FIFO

The author designed this dual clock FIFO, a block diagram shown in Figure 4.14 as a way for two circuits operating in different clock frequencies to communicate with each other, which is similar to the one that was used in KiloCore and KiloCore2. There is a read side and write side where data is stored into the internal memory of the FIFO using the write side clock and then read from the internal memory using the read side clock. This module is meant to be flexible, allowing to easily change the data width and address width as well as the size of the internal memory.

This project was motivated by the need for a simple dual clock FIFO similar to the dual clock FIFO designed by Ryan Apperson [93, 94] which is used inside of the VCL group's AsAP2 chip. To make this design Ryan's thesis was used as a guide, but the entire design was redone. It

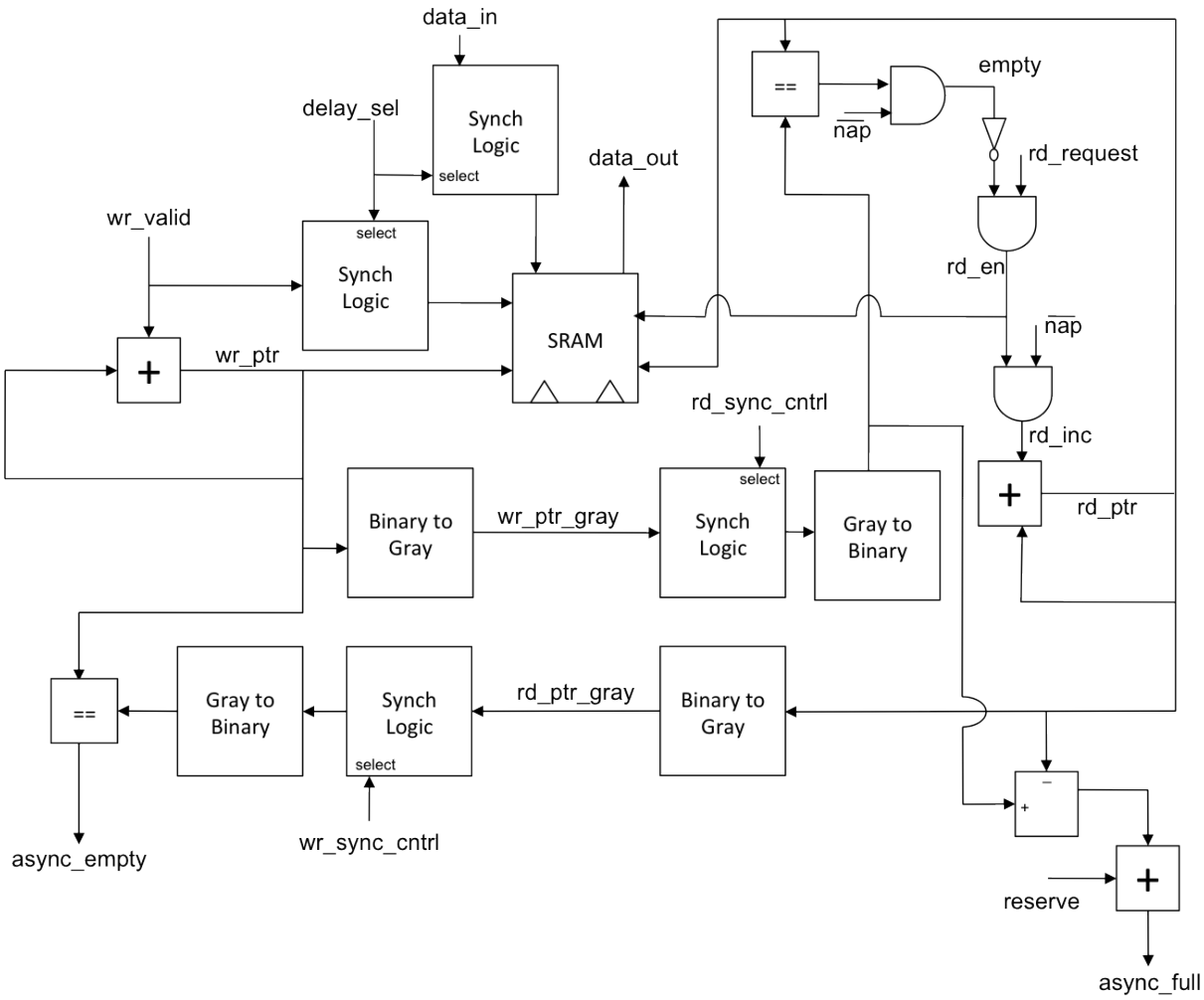


Figure 4.14: Block diagram of the designed dual clock FIFO.

was desirable to have a single module that was straightforward and easy to change.

The FIFO converts to gray code for sending address pointers across clock boundaries coupled with synchronization logic, shown in Figure 4.15 with a configurable amount of buffer registers that the pointer needs to go through, to mitigate metastability. As both sides operate on different clock frequencies, it is possible with a frequency disparity to write more data than is possible, because the slower side did not update fast enough. To remove this problem, without using power and area hungry handshaking, a configureable reserve space was implemented. When the total data stored plus the reserve space equals the total FIFO memory space, a full signal is given. This prevents the write side from sending too much data, at the cost of leaving a bubble of unused

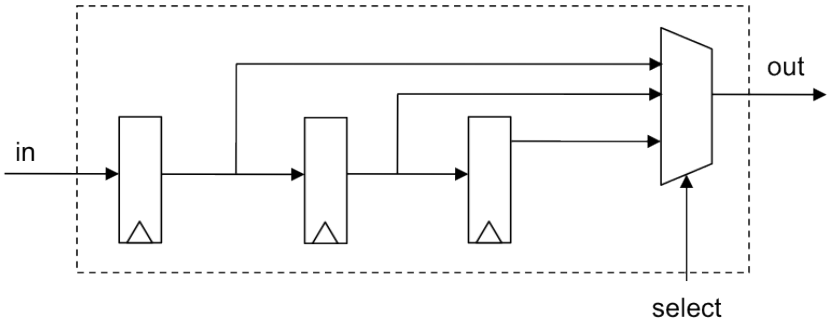


Figure 4.15: Block diagram of the configurable synchronization logic used in the dual clock FIFO.

space in the FIFO memories.

The design was tested with a testbench which will randomly choose a clock frequency for each clock after each clock pulse. Each side will randomly stop reading or writing accordingly for a random amount of clock pulses. The output is monitored and it will give an error if a non sequential output is given, which would mean an incorrect value was read.

4.5 Many-Core Physical Design Flow

As designing processor arrays in the 1000-processor era is a unexplored paradigm, it was necessary to develop a specific flow for the design of both KiloCore and KiloCore2. Throughout the flow, multiple programs were used from Cadence, Synopsys, and Mentor Graphics. The programs that were used are listed in Table 4.1.

Program Used	**Manufacturer**	**Purpose**
Design Compiler (DC)	Synopsys	Synthesis
Encounter (EDI)	Cadence	Place and Route
Virtuoso (IC)	Cadence	Final layout changes
Calibre	Mentor Graphics	DRC and LVS
UltraSim	Cadence	Full-Chip Simulator

Table 4.1: Programs use in the physical design process.

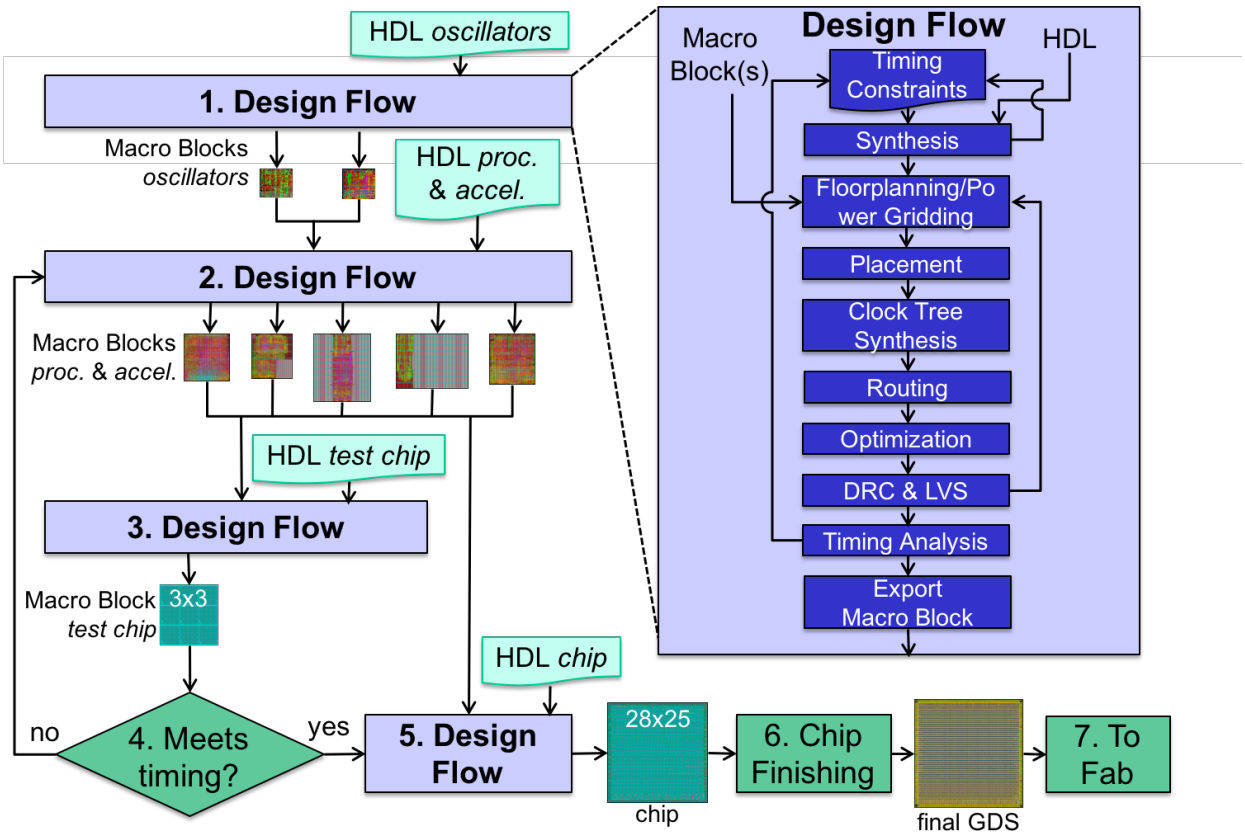


Figure 4.16: Illustration of many-core processor array physical design flow. KiloCore2 is used as an example.

4.5.1 Flow Steps

Both KiloCore and KiloCore2 had similar flows, with slight differences, manly attributed to the extra designs used in KiloCore2. Presented in this section is the flow devised for KiloCore2, but derivations of it could be used for any many-core design in the 1000-processor era with similar attributes to KiloCore2. An illustration of the physical design flow is shown in Figure 4.16. A detailed description of the flow is given below.

Detailed Flow

1. There is a need for 4 versions of a ring oscillator, designed from standard cells. So it is necessary to hand write gate-level netlist for the processor's oscillator, router's oscillator, high-speed processor's oscillator, and high-speed router's oscillator by selecting gates from the

standard cell databooks to create full custom ring oscillators along with other time sensitive parts. It is necessary to keep in mind both transition time of the selected gates, and the gate's drive strength, as wire capacitive loads can easily govern delay with small gate sizes.

2. There is still some control logic that is written in Verilog, so we need to synthesize these non-hand-selected parts for the processor's oscillator, router's oscillator, high-speed processor's oscillator, and high-speed router's oscillator in DC creating the 4 oscillator wrappers gate-level netlist files.

3. The wire-load inside of the ring oscillators are particularly important as they directly affect the speed at which it can perform. Therefore we place and route netlists for the processor's oscillator, router's oscillator, high-speed processor's oscillator, and high-speed router's oscillator. As these routings will not be used in the final design, it is just important to get a basic routing with similar characteristics to what will be in the final design. Clock tree synthesis is done on these oscillator wrappers at this stage as well to make sure that timing is considered in this closed unit. A gate-level netlist and a GDSII file need to be created of each for RC extraction.

4. Use the gate-level netlist and a GDSII files of the placed and routed processor's oscillator, router's oscillator, high-speed processor's oscillator, and high-speed router's oscillator in Calibre to extract a Spice netlist that contains all of the RC characteristics of the oscillator designs including the gates and wires.

5. Run a spice based simulation on the processor's oscillator, router's oscillator, high-speed processor's oscillator, and high-speed router's oscillator to test timing making sure to test all possible operating points to get a close estimate of how it will perform in silicon. It is a good practice to overshoot the estimated frequency so as to make sure unaccounted for physical differences, we have a full range of frequency coverage. As any change has multiple ramifications, it will be necessary repeat steps 1.–5. numerous times to get a good range of frequencies.

6. It was found that the performance could be increased by synthesizing these blocks separately from the rest of the processor: processor core, MAC, router, and clock pulse probing. Even though the oscillators are lower in the hierarchy than these blocks do not include them in the

synthesis, they will be added directly in the place and route stage, as those gates are already finalized, so we don't need synthesis touching them.

7. Use appropriate Verilog RTL along with the appropriate synthesis script folder to synthesize the processor, high-speed processor, Viterbi accelerator, FFT accelerator, and independent memory. This will synthesize the RTL into a gate-level netlist. These modules will call some already synthesized gate netlists.

8. Source the appropriate place and route scripts to place and route the generated gate-level netlists and SDC timing files for the processor, high-speed processor, Viterbi accelerator, FFT accelerator, and independent memory in EDI. Oscillators are added at this point, but are set to do not touch so that EDI does not try to resize any of these preselected gates. The oscillators are placed in a soft fenced region make sure the gates are close and minimal wires are routed above them. Generates a gate-level netlist and GDSII file to be used in extraction.

9. Use the previously generated gate-level netlist and GDSII files in Calbire to extract RC data from the processor, high-speed processor, Viterbi accelerator, FFT accelerator, and independent memory.

10. Run a spice based UltraSim simulation on the processor, high-speed processor, Viterbi accelerator, FFT accelerator, and independent memory to test timing, and repeat steps 6.–11. as necessary.

11. Use appropriate Calibre DRC decks with appropriate settings to check the GDSII files for design rule violations in the processor, high-speed processor, Viterbi accelerator, FFT accelerator, and independent memory. Use generated gate-level netlist and GDSII in Calibre to run LVS on them, making sure the layout matches the schematic. If there are any issues, go back to the previous step as appropriate, and repeat until both designs pass DRC and LVS.

12. Generate LEF, GDSII, and ILM files for the processor, high-speed processor, Viterbi accelerator, FFT accelerator, and independent memory for use at higher level hierarchies in EDI.

13. Synthesize the RTL for chip into a gate-level netlist with DC, without giving DC the netlists for the processor, high-speed processor, Viterbi accelerator, FFT accelerator, or independent

memory, using the same method mentioned in Step 6. This is the highest level, so it will include all of the drivers, glue, temperature sensor and I/O circuitry.

14. Place and route chip using the generated files LEF and GDSII from Step 13., using similar methods to Step 2.

15. Use static timing analysis to determine timing requirements for inter-processor communication. To get the timing information it will be necessary to use ILM information and a smaller test chip (as the computation time is rather large) to see if there are any timing or drive strength issues. Then go back to step 7. as necessary.

16. Generate gate-level netlist and GDSII files for chip for DRC and LVS on Calibre similar to the procedure in Step 12.

17. Generate LEF and GDSII of chip for use in IC chip finishing.

18. Some small last minute changes can be made in IC. IC generates final GDSII.

4.5.2 Found Problems and Solutions

In the process of creating the physical design of KiloCore and KiloCore2, a number of problems arose. A few of these problems, and their solutions, are listed below, as they can be helpful in the design of a chip in the 1000-processor era.

Problem 1a

Running power stripes over the hard macros was taking a very long time to complete, and never completed with a large array.

Problem 1b

In some instances the vias created were creating DRC errors with the vias that already existed in the processor and independent memory modules.

Solution 1

All the preexisting power rails were drawn at the highest level by a command that did not create vias or worry about DRC violations. The macros were placed on top of the wires, with blockages in place to stop any via creation over macros, as they already existed in the lower

level hierarchies. One needs to be careful with this solution, considering it makes no DRC check so one could accidentally create a illegal wire.

Problem 2

Wires between processors at the chip level were jogging on top of the processor macros, which lead to spacing DRC violations.

Solution 2

After testing it was determined that at the higher levels EDI was taking liberty in jogging signal wires over macros because there are explicit metal blockages in the LEF. For whatever reason, these blockages were not sufficient to stop a few of the long list of different metal closeness DRC violations. To fix this, in the lower level hierarchies of the processor and independent memory LEF files were exported without cut obstructions or any power rail information. This meant at the higher level of chip, it only knew where the signal pins were, and no other information. This forced it to take no chances when it was routing signal wires, making only straight wire connections, and ridding us of spacing DRC violations.

Problem 3

DRC violations about spacing between N-Well and P-Wells. They need to be either touching, or a minimum distance apart. If small filler cells were not placed in small gaps between standard cells, it would violate the rule. If macros were placed too close to each other, this could also show up. As EDI has no knowledge of the wells, it does not catch these violations.

Solution 3a

After talking about the two possible solutions (placing filler, or routing only as low as level 2), it was decided to just route down to level M2, so there was no issue with filler cells being blocked. This allowed for the final optimization step to take place, which couldn't have happened with the other option.

Solution 3b

To stop the violation between macros, a halo had to be added of sufficient size, which lead to the macros not being perfectly abutted, another option could have been to exactly abut them, but then the designer would have to worry about metal spacing violations in the lower level

hierarchies, because at higher levels there is no information on the lower hierarchy metals given.

Problem 4

EDI did not automatically route larger metal connections between C4 bumps and the power grid.

Solution 4

A script was written to manually draw metal shapes to connect the bumps with multiple wires, being careful not to go over the metal density design rule.

Problem 5

The power gates were not being connected to their corresponding power rails. As far as could be determined, EDI is not designed to use power gates the way that we used them to switch between power rails.

Solution 5

When the power gates are placed, a prefix was given to them that was specific to the power rail they were supposed to connect to, and used a command to brute force the cells to connect to their respective power rails.

Problem 6

There were multiple issues getting the level shifters to connect correctly to the always-on power grid.

Solution 6

In the layout script, EDI was told that the level shifter power pins were nets, and routed with a non-default rules telling it to make these wires wider and not to branch out as many signals from it.

Problem 7

Because of Solution 6, we got DRC violations because there were large metal wires that didn't have the required number of vias.

Solution 7

The non-default rule mentioned in Solution 6 was modified to include special rules for force extra vias when doing this particular routing.

Chapter 5

Design and Results of GALS Processor Array Chips

5.1 KiloCore

The first fabricated 1000-processor integrated circuit is presented [95], shown in Figure 5.1. At maximum supply voltage, preliminary results show that processors operate up to an average maximum clock frequency of 1.78 GHz; therefore, one chip can execute a maximum of 1.78 trillion operations per second. At a reduced supply voltage, one chip can execute a maximum of 1 trillion operations per second while dissipating 13.1 W. At its lowest operating voltage, 0.56 V, it can perform 5.8 pJ/Op while performing 115 Billion Ops/sec.

Our 1000-processor chip serves as an important milestone in the progression of digital processor design [1, 12]. The 32 nm PD-SOI CMOS KiloCore integrated circuit contains 1000 MIMD programmable processors and 12 independent memory blocks containing a total of 768 KB, where each memory block may be used simultaneously for data and instructions, as shown in Figure 5.2. The 1000 processors are arrayed in 32 columns and 31 rows with eight processors and 12 independent memories in a 32^{nd} row, shown in Figure 5.3. The processors and memories operate more like computers within a datacenter than like a traditional multi-core chip.

While caches are generally extremely effective, they are problematic for resources such as chip I/O and cache-coherency circuits when the number of processors is scaled into the 100s or 1000s and they also dissipate significant power [2]. In contrast, KiloCore processors do not contain explicit

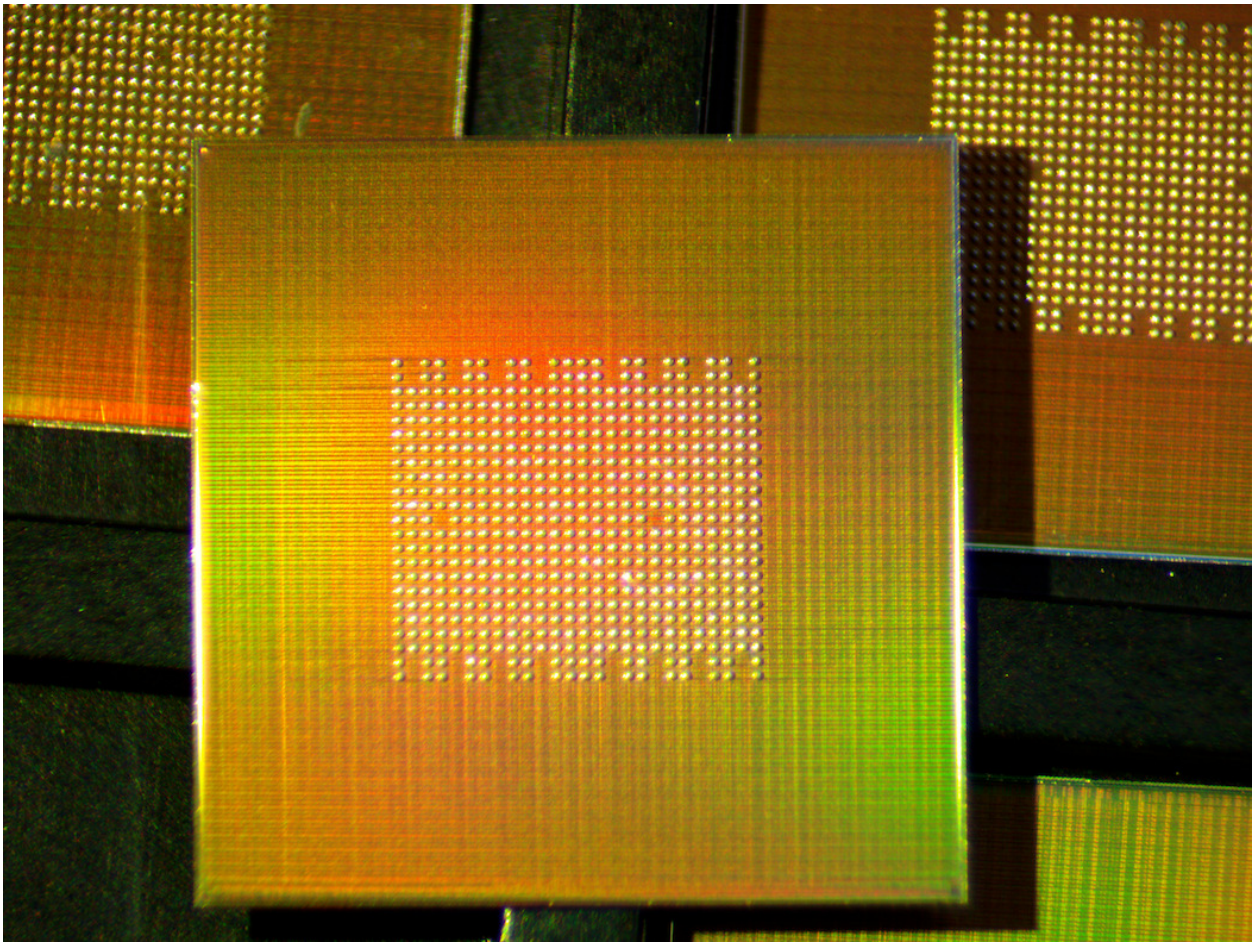


Figure 5.1: Photograph of bare die KiloCore chips.

caches and instead store data and instructions inside i) local memory, ii) an arbitrary number of nearby processors, iii) independent on-chip memory blocks, or iv) off- chip memory. If desired, caches or other memory schemes can be constructed by programmer-specified software running on dedicated processors.

KiloCore's 1000 processors, 1000 packet routers, and 12 independent memories are clocked by local and completely-unconstrained (below the maximum operating frequency) clock oscillators that do not use PLLs and may change frequency, halt within 1-5 clock period, and restart in less than one clock period to reduce power dissipation. Processors with no work to do dissipate exactly zero active power (leakage only) which is an important capability in many-core chips due to the increasing prevalence of processors that are not always active. At 0.9 V, idle processors leak 1.1%

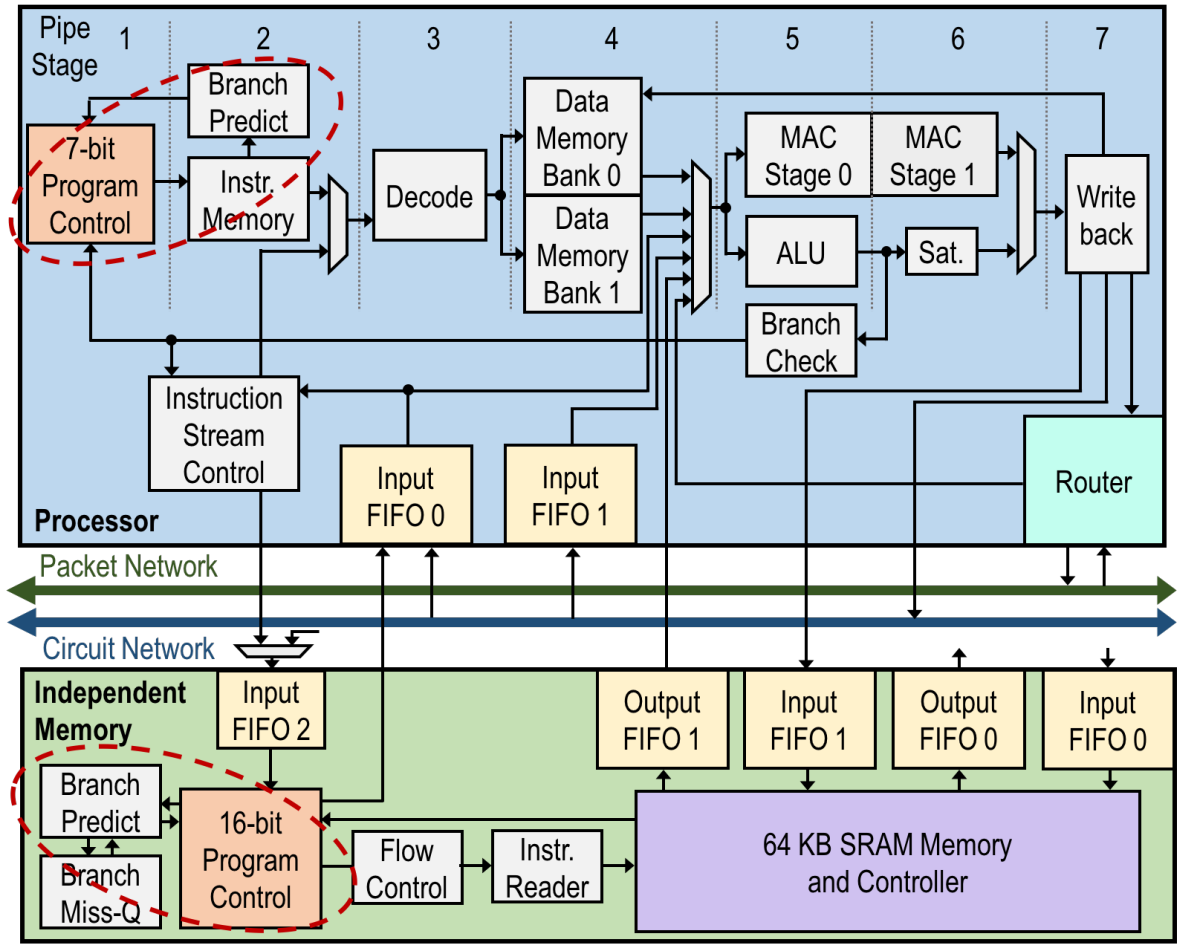


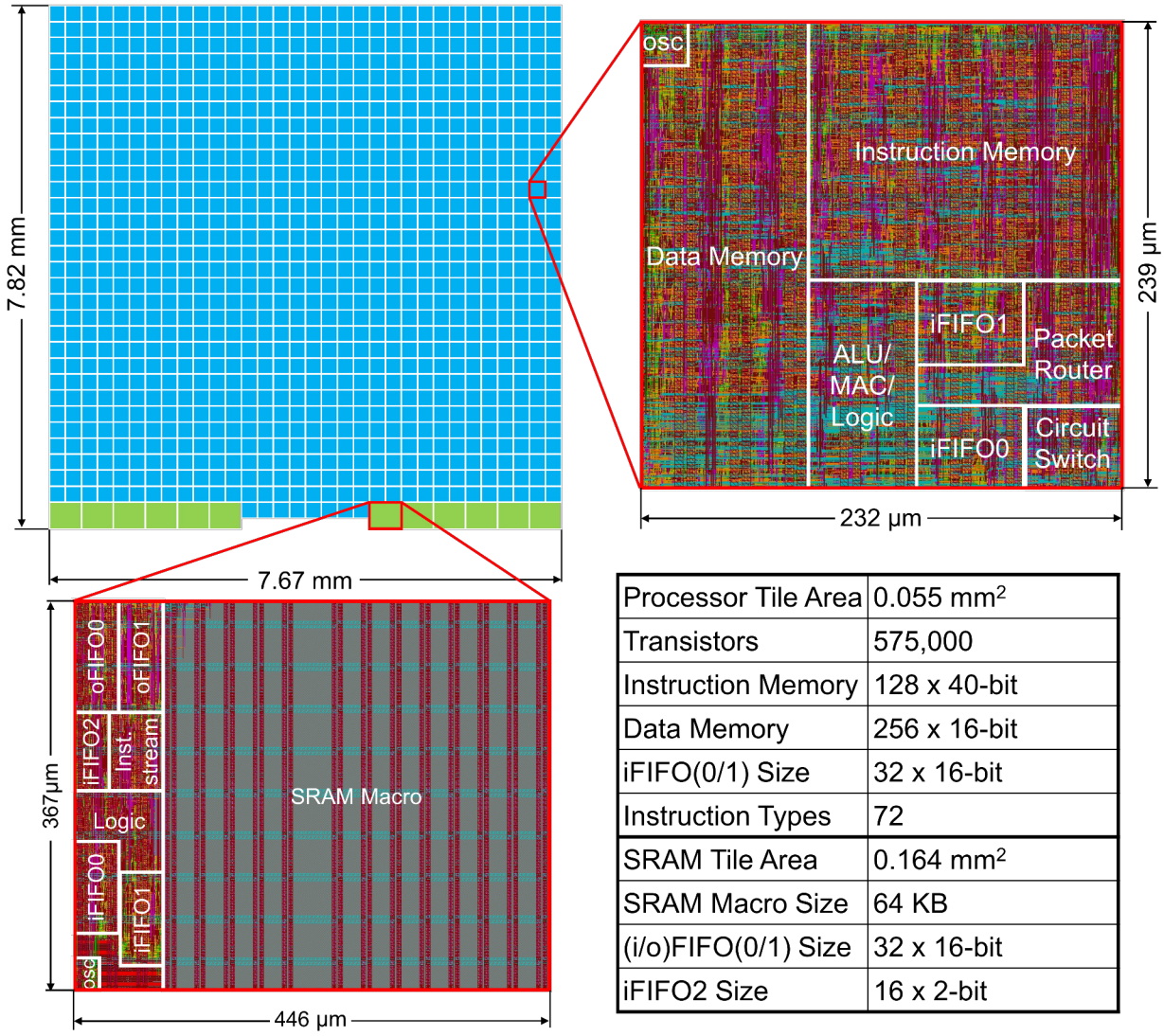
Figure 5.2: Pipeline diagram of a KiloCore processor and adjacent independent memory with program control highlighted in each block.

of their typical operating power. Information reliably crosses unrelated clock domains through dual-clock FIFO buffers.

5.1.1 Design

The KiloCore computational system consists of 1000 independent, uniform, programmable, RISC-type, in-order, single-issue processors; 1000 packet routing modules; and 12 shared memory modules, as shown in Figure 5.4. Each processor, packet router, and shared memory module contains its own clock oscillator, which is halted when the module is idle.

Each processor contains a 128×40-bit instruction memory, 256×16-bit of data memory, three data address generators, two 32×16-bit input dual-clock FIFO buffers, a 16-bit fixed-point



Processor Tile Area	0.055 mm^2
Transistors	575,000
Instruction Memory	128 x 40-bit
Data Memory	256 x 16-bit
iFIFO(0/1) Size	32 x 16-bit
Instruction Types	72
SRAM Tile Area	0.164 mm^2
SRAM Macro Size	64 KB
(i/o)FIFO(0/1) Size	32 x 16-bit
iFIFO2 Size	16 x 2-bit

Figure 5.3: The KiloCore chip, including a block diagram on the top left, a plot of one processor with major components highlighted on the top right, a plot of one 512-Kbit independent memory block in the bottom left, and details about the submodules.

data path, and support 72 instructions. A plot of the layout of the single processor is shown in Figure 5.5, and the area breakdown is shown in Figure 5.6, corresponding to the locations given in Figure 5.7. The instruction set includes signed and unsigned operations to enable efficient scaling to 32-bit or larger word widths. Processors support conditional execution and static branch prediction. Compile-time application profiling optimizes branch paths and minimizes misprediction penalties. Although the natural word width of the datapaths and memories is 16 bits, the multiply-accumulator

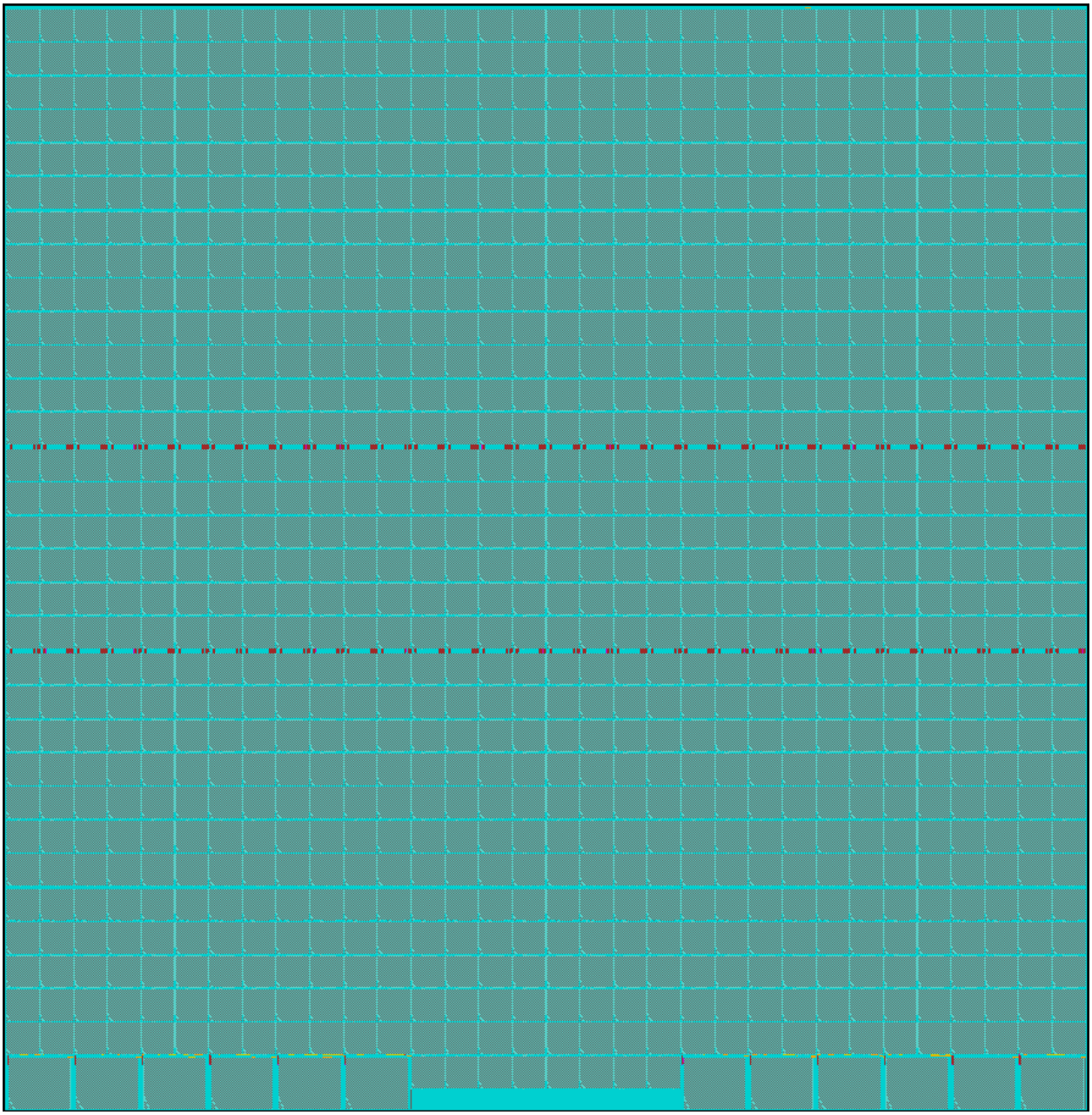


Figure 5.4: A plot from Encounter showing all submodules inside the KiloCore array. Notice the 12 memory modules on the bottom, and the 1000 processors in the remaining space. This plot shows just the array, without any of the peripheral I/O drivers, ESD modules, or capacitors, which can be seen in Figure 4.4.

has a 40-bit output and through software, other word widths are easily handled, for example, 32-bit floating point [73] and 10-Byte sorting keys for 100-Byte data records [46].

Communication on-chip is accomplished by two complementary means: a very high-

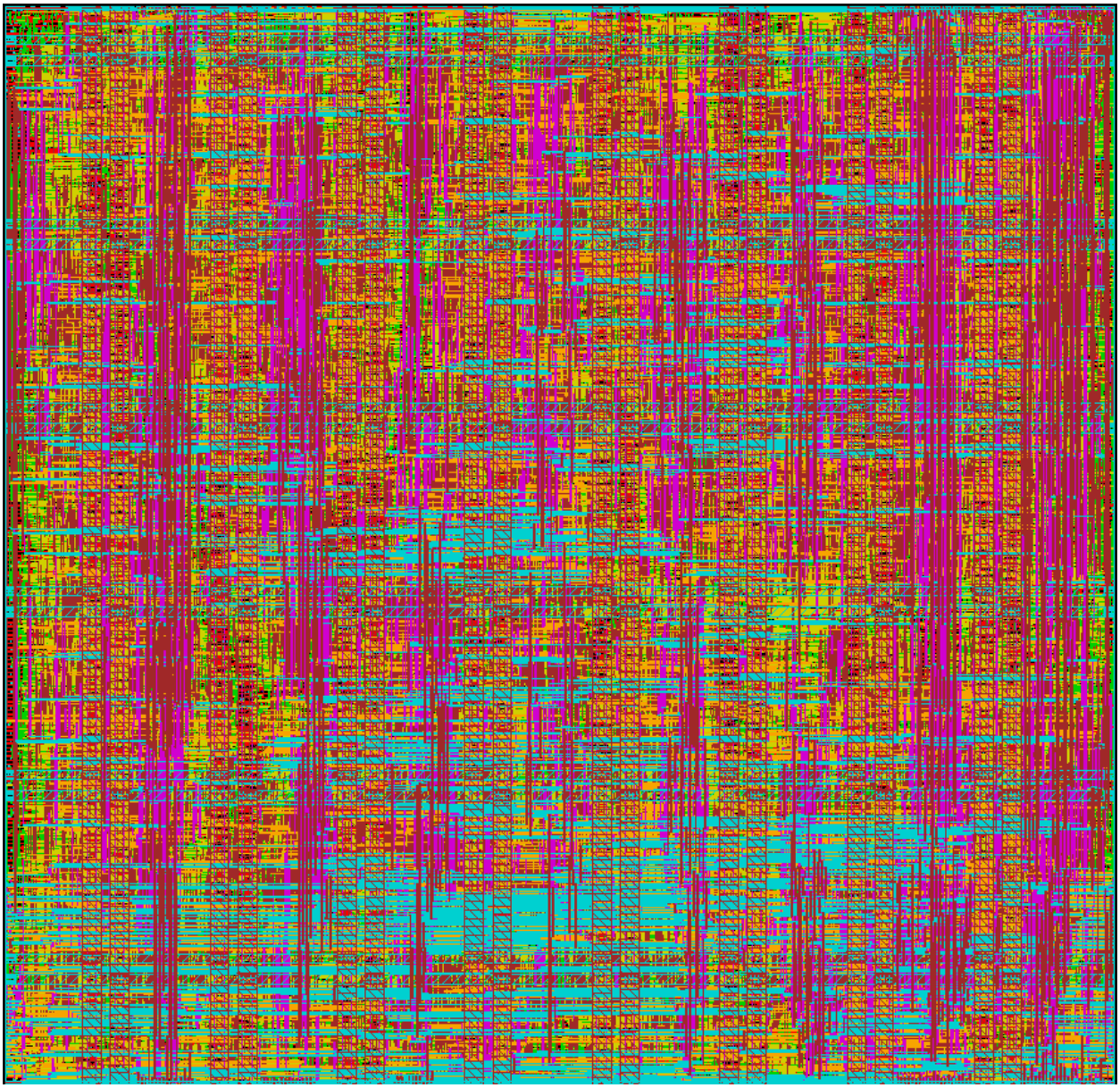


Figure 5.5: A plot from Encounter showing the placed and routed KiloCore processor tile. All wires are visible, including power, ground, and signal wires.

throughput circuit-switched network [53] and a very-small-area packet router [96], highlighted in Figure 4.13. The circuit-switched links are source-synchronous, meaning 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 31.1 Gbits/sec. Each of the four edges of each processor

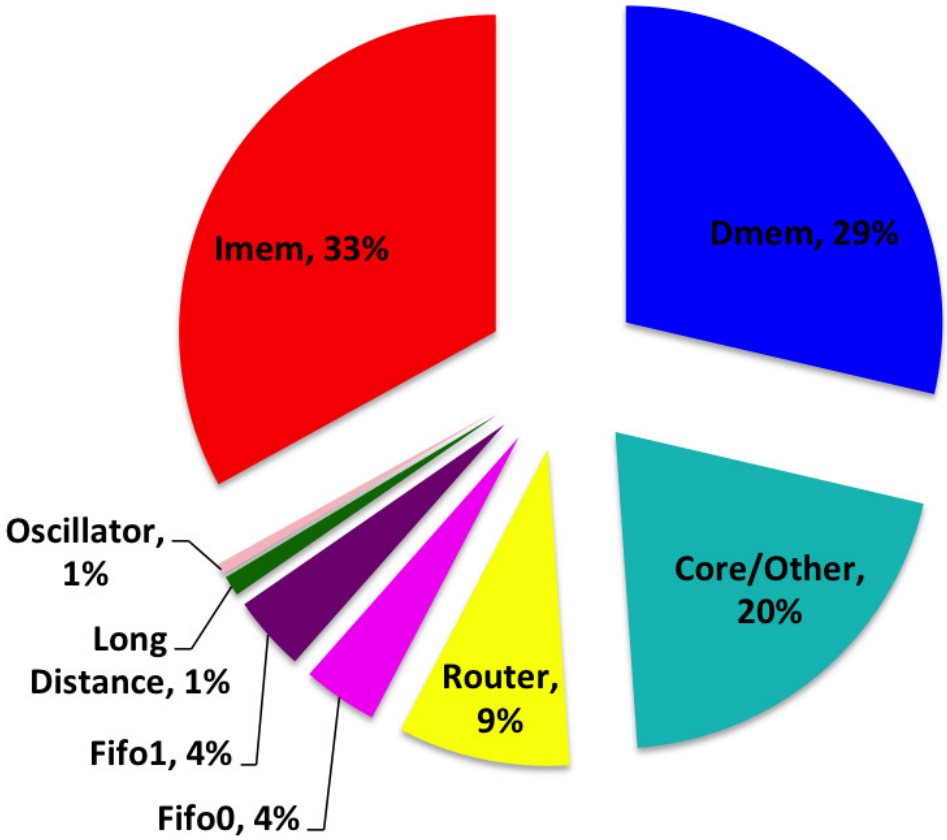


Figure 5.6: A pie graph showing the area breakdown of a KiloCore processor tile. The colors in the pie graph correspond to the colored standard cells in Figure 5.7.

has two such links entering and two links exiting the processor. The packet router inside each processor occupies only 9% of each processors area and is especially effective for high fan-in (many to one) and high fan-out (one to many) communication. Each router supports 38.2 Gbits/sec of throughput with a maximum of 7.6 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. Both network types contribute to a total bisection bandwidth of 4.4 Tbits/sec.

Independent memory modules each contain a 32,768×16-bit static random-access (SRAM) memory which is shared between two neighboring processors, as shown in Figure 5.8. Each module contains two 32×18-bit input buffers, two 32×16-bit output buffers, and one 16×2-bit processor response buffer. Memories support random and burst access for data reading and writing, and are capable of streaming instructions using a dedicated control module, as shown in Figure 5.2.

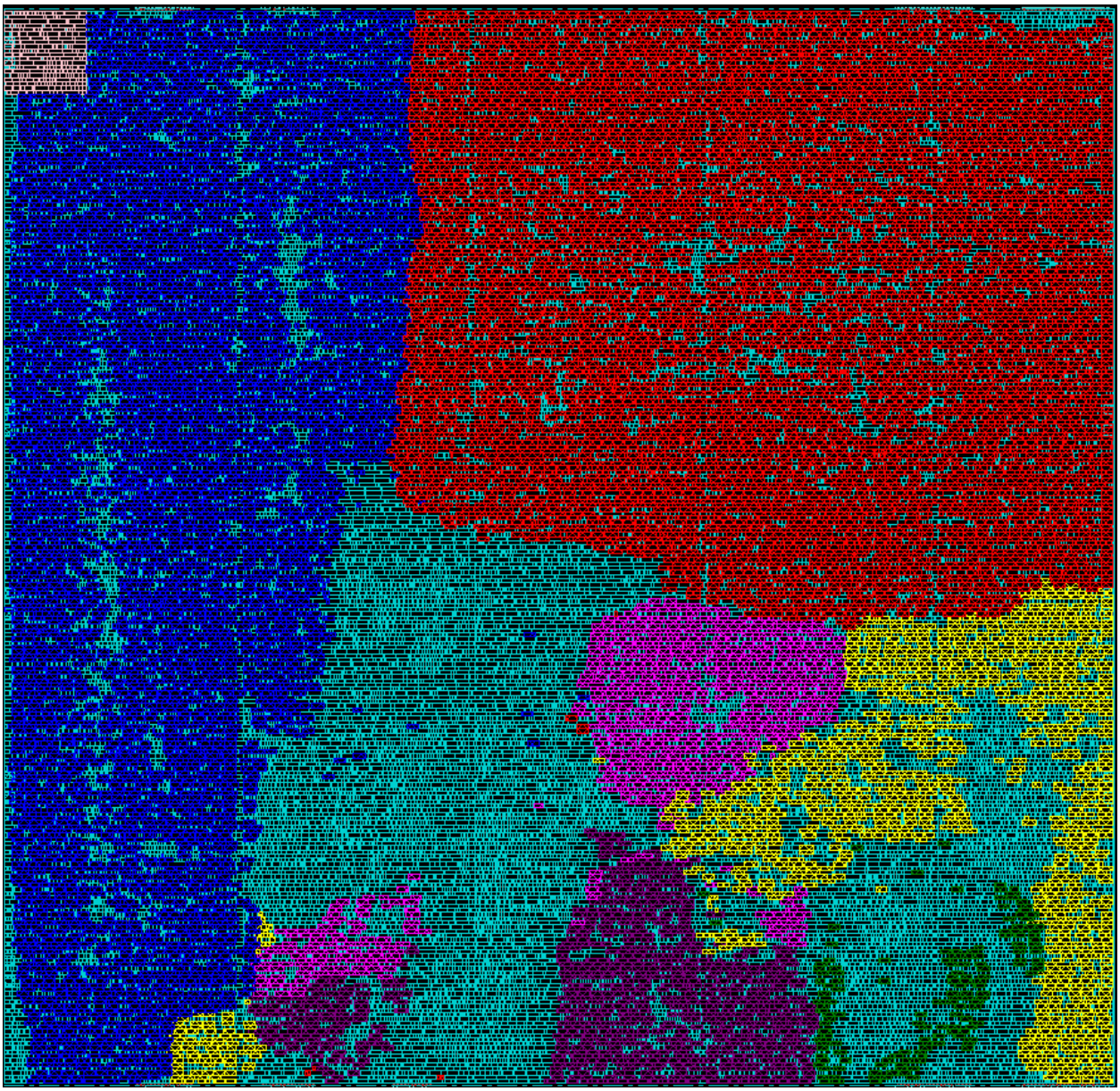


Figure 5.7: A plot from Encounter showing a KiloCore processor tile, highlighting the used standard cells by different functional portions of the tile. The names and percentages of each of these sections are shown in Figure 5.6.

Each processor issues one in-order instruction per cycle into its 7-stage pipeline from its 128×40-bit instruction memory, as shown in Figure 5.2. None of the 72 supported instruction types are algorithm-specific, which has two interesting results: in general, the chip compares better to other processors while computing more complex benchmarks than simpler ones, and secondly, it efficiently computes applications from a wide variety of domains. Instruction input operands

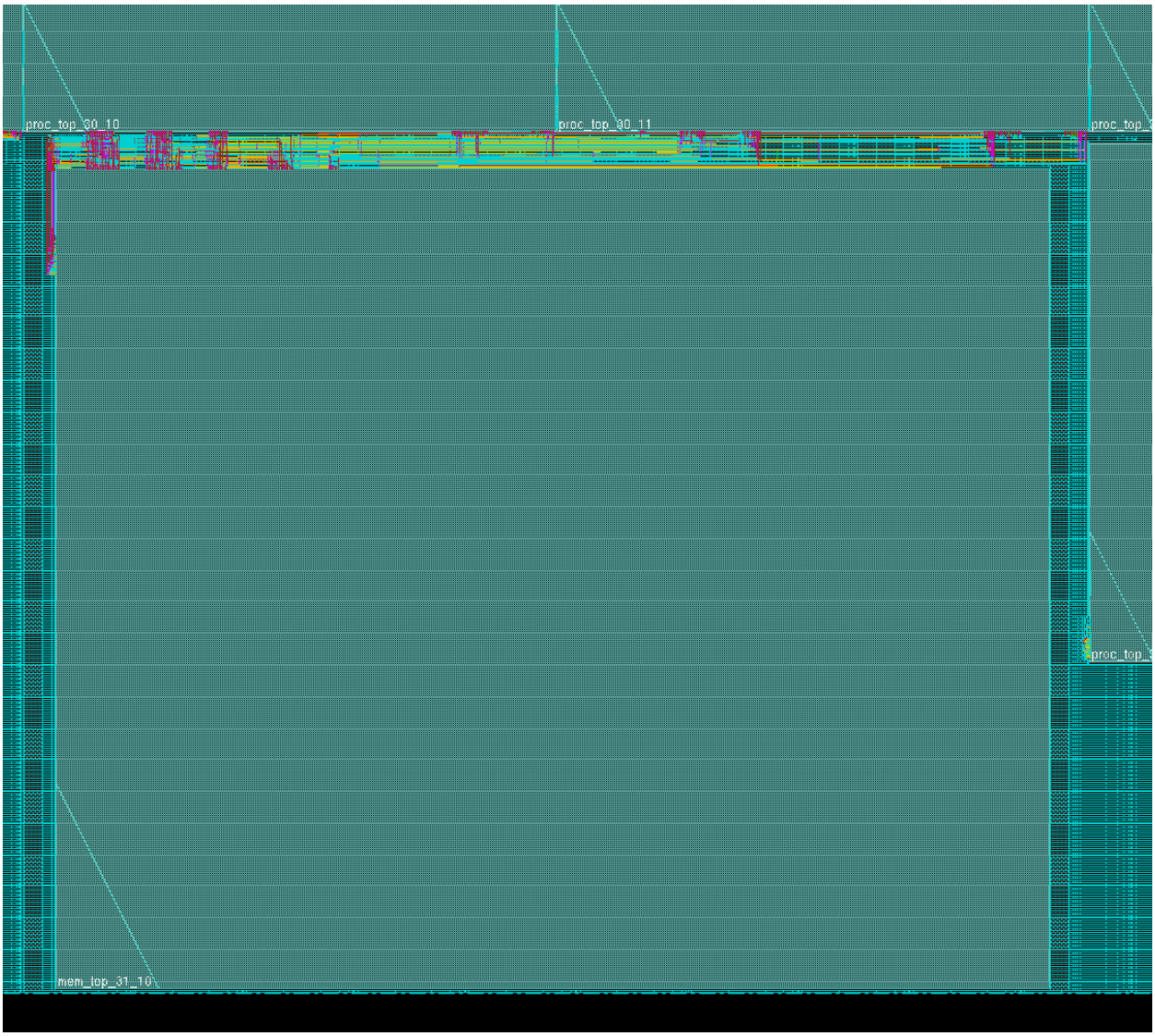


Figure 5.8: A plot from Encounter showing the connection in the KiloCore array between the processors and an independent memory. Each independent memory is connected to the two processors directly above it. Also shown is the beginning of the 32$^{\text{nd}}$ row of processors on the right side of the plot.

and output results may come from or go to the local data memory, one of several circuit-switched network ports, the packet router port, an attached large memory block, or a few other registers.

The data memory is implemented as two 128×16-bit banks to sustain a throughput of one instruction per cycle with common instructions that require one destination and two source operands, shown in Figure 5.9. However, profiled code of five disparate applications showed that only 0.34% of operands could not be mapped to an address in only one bank and thus needed to be

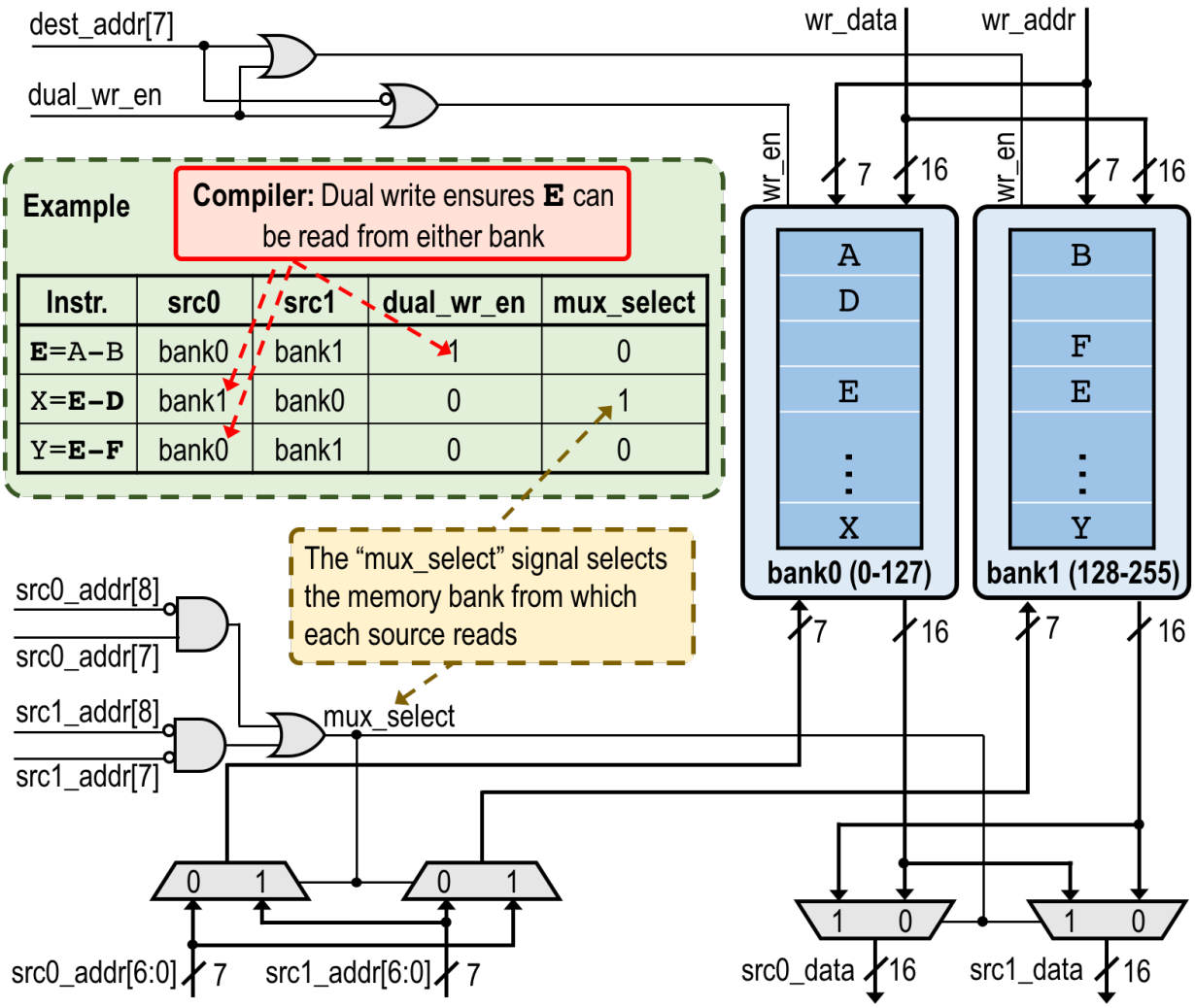


Figure 5.9: A circuit diagram showing the dual memory bank implementation of DMEM and how it reads and writes back in KiloCore.

written to both banks redundantly to avoid conflicts during subsequent reads. Our scheme permits conflict-free addressing with optimal memory space maximization. Hardware is controlled by MSB address bits for reads and the opcode to determine whether writes are to a single bank or both.

The high-throughput circuit-switched network is especially efficient execution of an instruction with an input operand transferred from an adjacent processor dissipates only 11% additional energy compared to if it were transferred from the local data memory. An additional 47% beyond the energy for a local access is needed if the operand comes from a processor ten processors away. Information reliably crosses unrelated clock domains through dual-clock FIFO memory buffers [94].

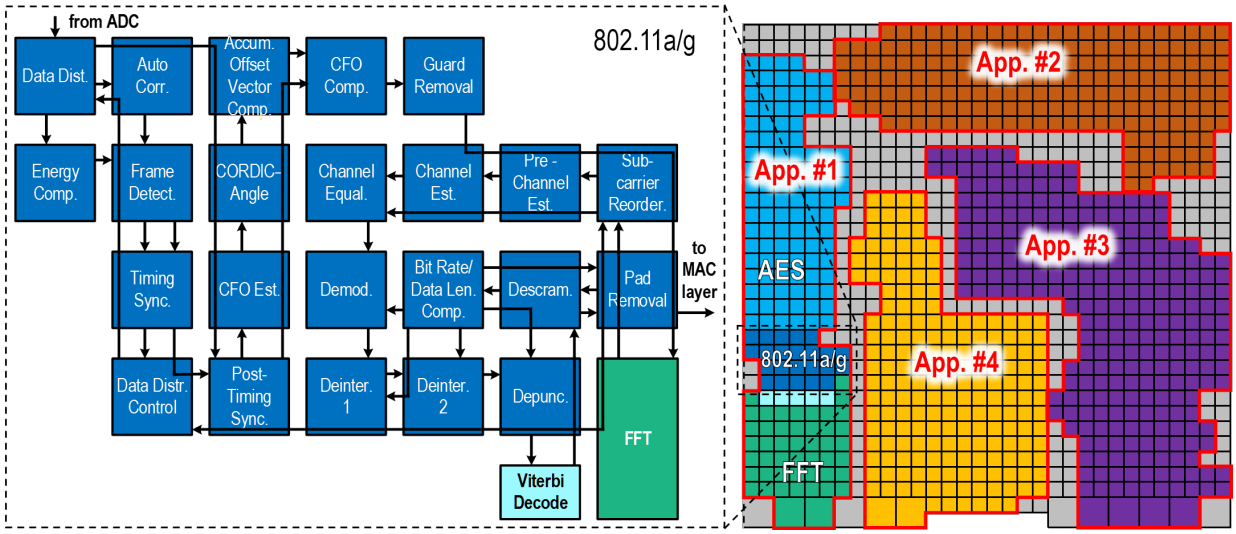


Figure 5.10: Application mapping example. An automatic mapping tool generates an optimized mapping solution for the application and targeted chip region. Multiple applications can work together or separately. In this example, App. #1 contains an AES encryption block and an 802.11a/g Wi-Fi receiver which utilizes an FFT and a Viterbi decoder. Apps. #2-4 are mapped into other cores and simultaneously compute other workloads.

5.1.2 Programming and Application Mapping

Programming is accomplished by a multi-step process, as displayed in Figure 5.10. The application mapping tool is intended to be run by an attached administrative processor that may calculate new mappings during runtime for purposes such as: simultaneous execution of unrelated workloads, optimizing mappings to take advantage of process, voltage, temperature (PVT) variations, avoiding faulty or partially-functional processors, or for self-healing for failures due to wear-out effects such as negative bias temperature instability (NBTI), hot carrier injection, or electromigration [24, 97, 98]. Several dozen digital signal processing (DSP) and general tasks have been coded plus more complex applications including: families of FFT processors, JPEG encoders, Viterbi decoders, families of advanced encryption standard (AES) encryption engines [80], a fully-compliant IEEE 802.11a/11g Wi-Fi wireless LAN baseband transmitter and receiver [99], a large portion of the mid- and back-end processing for a medical ultrasound unit [100], a synthetic aperture radar (SAR) engine [74], and a complete scalable H.264 HDTV encoder including a 1080p 30 frames/sec residual encoder [81]. Power, throughput, and area results compare very well with solutions on other processors. Recent projects implemented sorting [46] and pattern matching [86]

of arbitrarily large databases of benchmark 100-Byte data records, which also has applications in network packet inspection and processing. Another recent project implemented single-precision floating point in software and presents an interesting possibility for applications that require high performance in both floating-point and integer domains [73].

5.1.3 Measured Results from KiloCore

Measurement Methodology

Chip programming and testing is carried out on the host PC using a set of software libraries written in the Python programming language. Tests were designed to verify functionality, measure frequency, and measure energy per operation across a range of voltages. Each processor was checked for operational correctness by performing a test sweep of the instruction set, instruction memory, and data memories. Application mapping tools may be configured to avoid faulty processors found on any given chip.

Frequencies are measured for each processor by testing the maximum speed of the critical path in its logic. Tests are performed on one processor at a time, with other processors placed into a low power standby state, to minimize the impact of power delivery limitations in the test chips low-cost packaging solution. The maximum operating frequencies range from 1.827 to 2.068 GHz with a supply voltage of 1.05 V, with a mean of 1.947 GHz. Multiply-accumulate (MAC) operations are designed to utilize two processor cycles in normal operation, but may be performed in one cycle at frequencies up to 1.19 GHz.

The test chips BGA package that was available to us was designed for another chip and is capable of directly powering the central 160 processors in the array, see Section 5.1.4 for more details. Other processors experience reduced power delivery and a matching reduction in maximum frequency, proportional to the number of actively running processors in their region due to resistance in the on-chip power grid. A maximum of 1.95 trillion operations per second is achievable with a custom chip package capable of delivering power uniformly across the array and this is reported.

The energy used by each operation is calculated using measurements from a series of paired test programs, where the two programs will differ only by whether they include a given operation. Tests are performed using 50 processors in the center of the array each running an

instance of the program, reducing the effects of measurement inaccuracy at low processor counts while avoiding power delivery limitations in the chip packaging. Inter-core communication tests utilize 100 processors transferring data through 50 links. Measurements are taken across a range of frequencies to verify energy per operation remains constant. Tests use fully randomized data to capture the worst case energy requirements; a typical application is expected to use less energy as most operations do not exercise the full data width.

At the minimum supply voltage of 0.75 V, single cycle instructions consume between 5.4 pJ and 8.6 pJ. Local memory reads and writes consume 0.78 pJ and 1.98 pJ respectively. Circuit network communication requires 4.2 pJ for the first inter-core hop (including FIFO activity), and an additional 0.36 pJ or 0.51 pJ per subsequent hop, depending on if a register buffer is included if desired or to increase data integrity.

Average energy per operation, 10.6 pJ at 0.75 V and 21.6 pJ at 1.05 V, is calculated using a weighted average based on the profile of a simulated fast Fourier transform (FFT) application utilizing 327 processors. The average FFT operation includes 0.77 memory reads, 0.43 memory writes, and 0.33 inter-processor data transfers. Multiply-accumulates (13% of operations) and mispredicted branches (0.1% of operations) require multiple cycles to complete and include the related energy for these cycles.

Bisection bandwidth, 4.47 Tbps, is calculated for 32 rows down the center of the array, where each row includes two circuit links and one packet link in each direction. A circuit links bandwidth, 31.1 Gbps, is based on transferring 16 data bits each cycle at 1.947 GHz. A packet routers bandwidth, 7.6 Gbps, is based on a measured 477 million packets transferred each second at the highest functional router clock frequency, with 16 data bits in each packet.

Processor Array Performance

Three critical metrics for digital processors are performance (both throughput and latency), energy per operation (which equals power at a specified workload rate), and circuit area (because of its impact on cost and its obvious link to how much functionality can be placed on each chip) [103]. Processor maximum operating frequencies range from 1.70 GHz to 1.87 GHz with an average of 1.78 GHz at 1.10 V, which implies a maximum execution rate of 1.78 trillion MIMD operations per second for each chip, as shown in Figure 5.11 and Table 5.1. At a supply voltage of 0.84 V, 1000

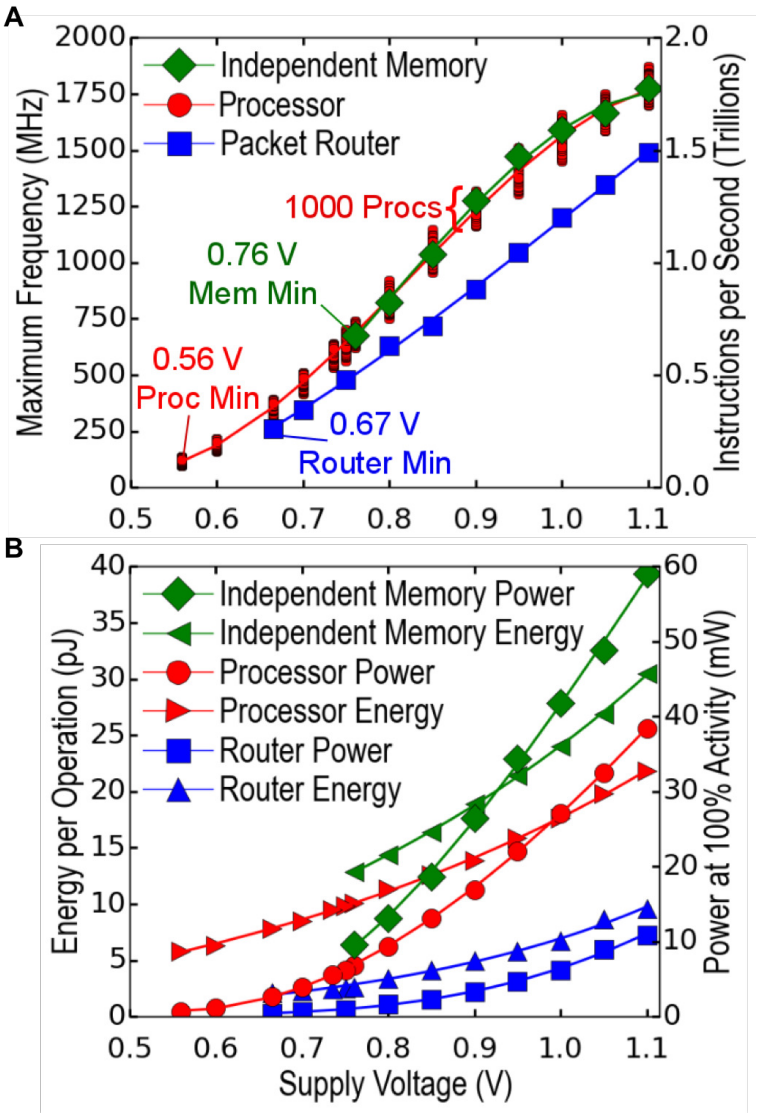


Figure 5.11: Preliminary KiloCore measured data at various supply voltages. (A) Maximum operating clock frequency and maximum number of instructions per second. (B) Energy dissipated per instruction and power dissipation at maximum clock rate using an instruction mix from an FFT application.

active cores could process a maximum of 1.0 trillion operations each second while dissipating 13.1 W. At a supply voltage of 0.56 V, processors dissipate 5.8 pJ per operation at 115 MHz, which would enable a chip to process 115 billion operations per second while dissipating only 1.3 W. Processors achieve their optimal energy times time of 11.1 pJ×ns at a voltage of 0.9 V. Independent memories operate from 1.77 GHz at 1.1 V down to 675 MHz at 760 mV. Routers operate from 1.49 GHz at 1.1 V down to 262 MHz at 665 mV. 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 likely lowers the minimum operating voltage.

Table 5.1: Preliminary results from KiloCore compared to other chips at various supply voltages.

	Processor Count	Technology (nm)	Clock Frequency (MHz)	Supply Voltage (V)	Per Processor Power (mW)	Energy/Op (pJ)	E×T (pJ×ns)
Sleepwalker [101]	1	65	25	0.4	0.065*	2.6	104
			23.6	0.375	**0.052***	**2.2**	93.2
TeraFlops [21]	80	65	**4000**	1.2	2260**	70.6***	17.7
			3130	1	1230**	49.1***	15.7
AsAP2 [18]	167	65	1070	1.2	47.5	44	41.1
			66	0.675	0.61	9.2	139
Am2045 [102]	336	130	300	-	23.8*	79.4	265
KiloCore *This work*	**1000**	32	1782	1.1	39.6	21.9	12.2
			1237	0.9	17.7	13.8	**11.1**
			638	0.75	6.9	9.9	15.4
			115	0.56	1.3	5.8	50.3

* Per processor power calculated by multiplying energy per operation by clock frequency

** Per processor power calculated by dividing total power by number of cores

*** Energy calculated by dividing power by clock frequency and IPC (assumes processor is fully active for every clock cycle)

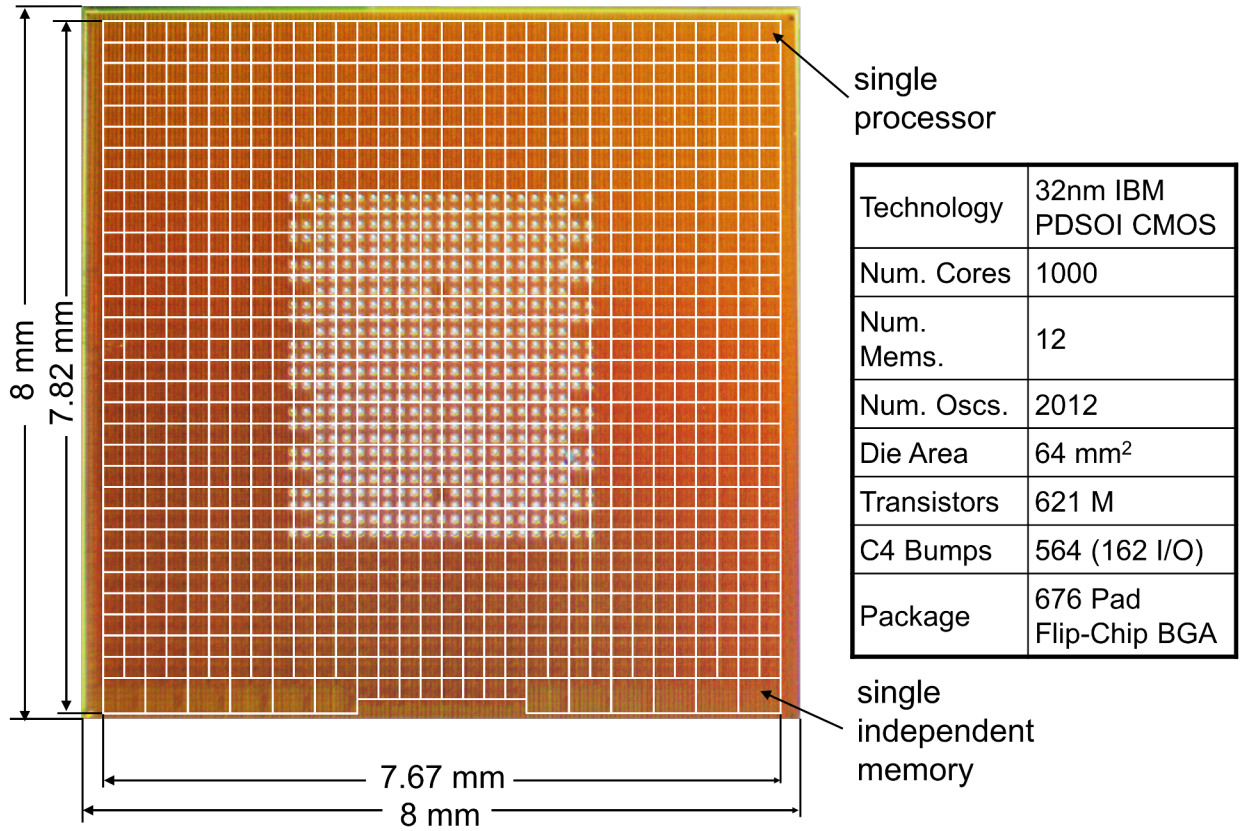


Figure 5.12: Micrograph of the KiloCore chip, with key statistics.

5.1.4 Chip and Test Setup

Chip

The KiloCore chip was fabricated in a 32 nm CMOS technology. Each processor contains 575,000 transistors and occupies 239 µm by 232 µm; therefore 18 processors occupy 1 mm^2. The entire array is 7940 µm by 7940 µm and contains 621 million transistors. The array area, which includes the 1000 processors and 12 shared memories measures 7667.32 µm by 7820 µm. The complete chip is 8 mm by 8 mm, as shown in Figure 5.12.

Package

Data I/O and power delivery are supported using 564 C4 solder bumps near the center of the array to connect (Figure 5.12) to a 676-pin flip-chip ball grid array (BGA) package (Figure 5.13).

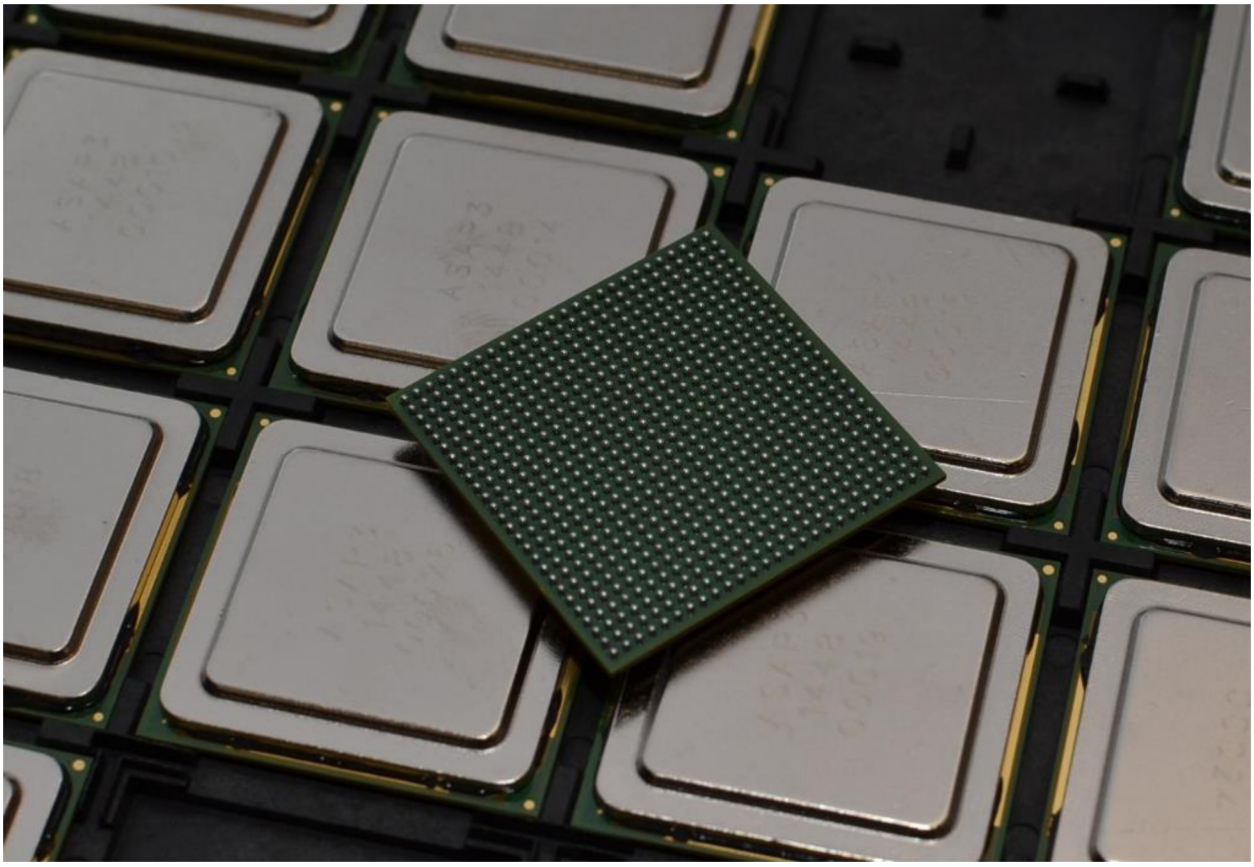


Figure 5.13: Packaged KiloCore chips, showing the top and bottom. Close-up of a tray of packaged KiloCore chips. The underside of the BGA package shows the solder balls that attach KiloCore to the custom daughtercard.

I/O signaling is handled by 64 low-voltage differential signaling (LVDS) drivers and 38 single-ended drivers. Drivers are placed along the periphery of the processor array. Ten analog voltage probe points are included to support accurate on-chip voltage measurements. An image of a package interfaced with a daughter card is shown in Figure 5.14.

Test Setup

The KiloCore PCB attaches to a commercial Xilinx VC709 FPGA board, as shown in Figure 5.15. The FPGA is used to interface between the KiloCore daughtercard and the desktop PC. Data buffers for KiloCore's high-speed data ports are implemented as FIFO buffers.

The laboratory test station for KiloCore, shown in Figure 5.16, includes the KiloCore PCB daughtercard, a commercial FPGA board, a suitable desktop computer, and numerous laboratory

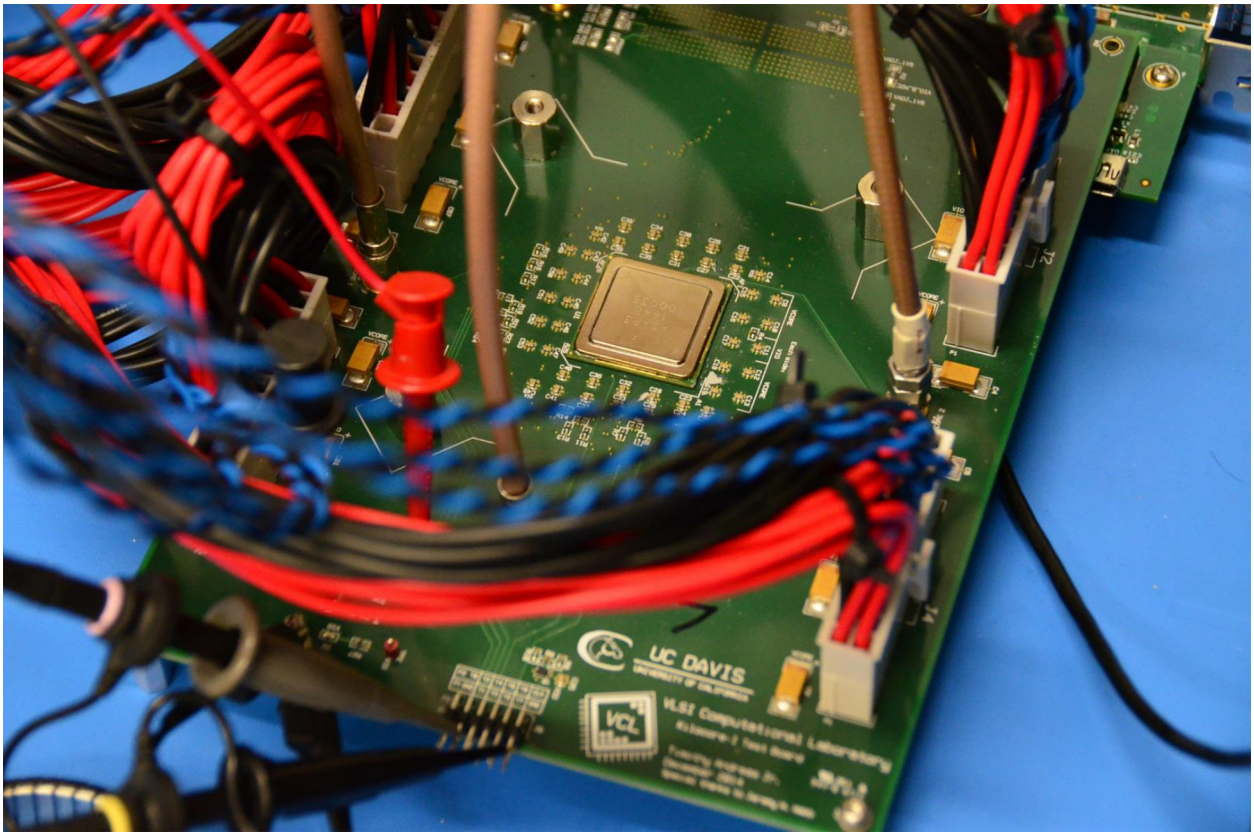


Figure 5.14: KiloCore daughtercard, showing a mounted package. Close-up of the custom daughter card, with the KiloCore BGA package in the center.

instruments such as power supplies, multimeters, and oscilloscopes.

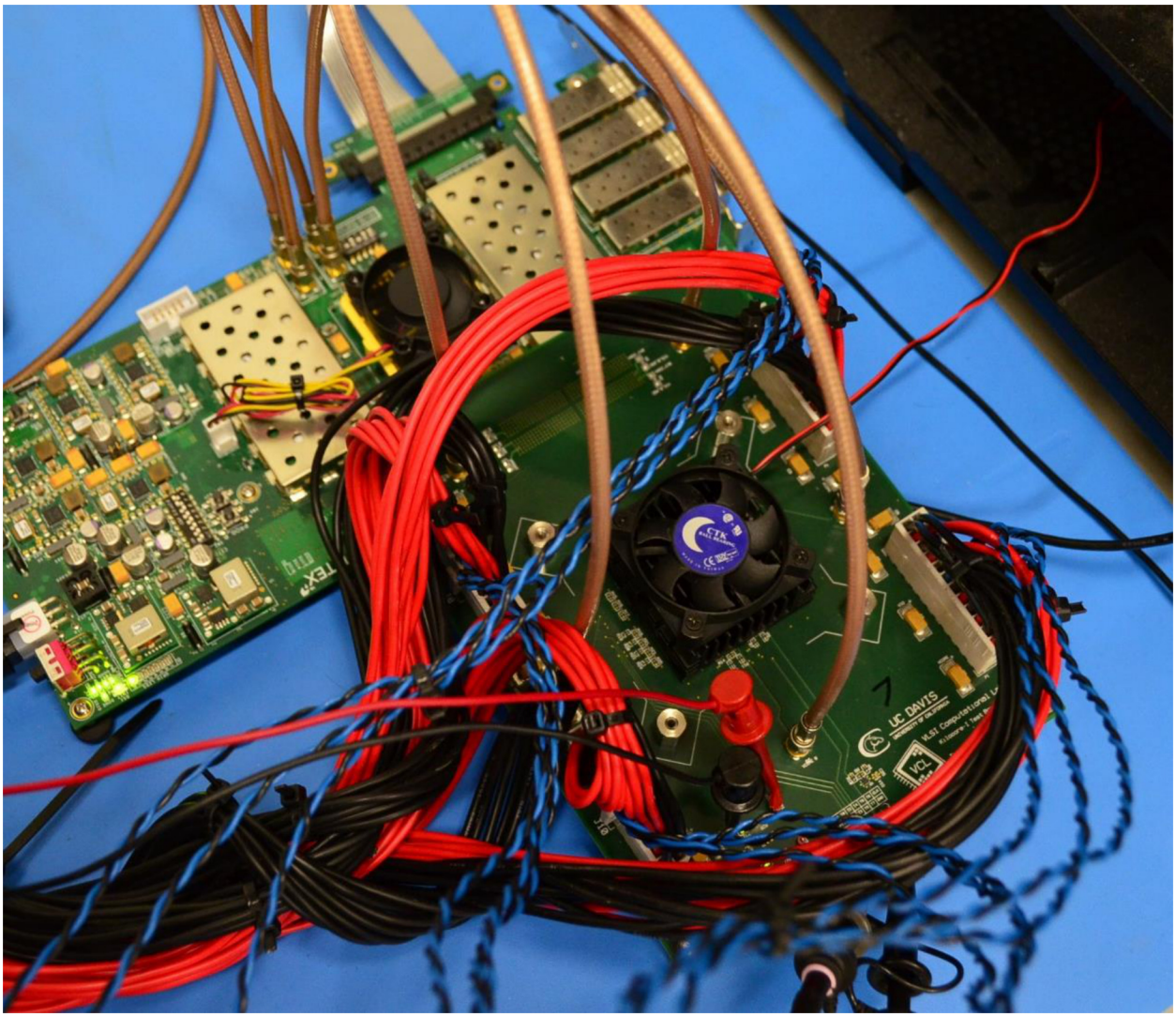


Figure 5.15: Programming setup with a close-up view of the FPGA board (upper left) and KiloCore printed circuit board daughercard (lower right) with a heat sink and fan attached to the KiloCore chip.

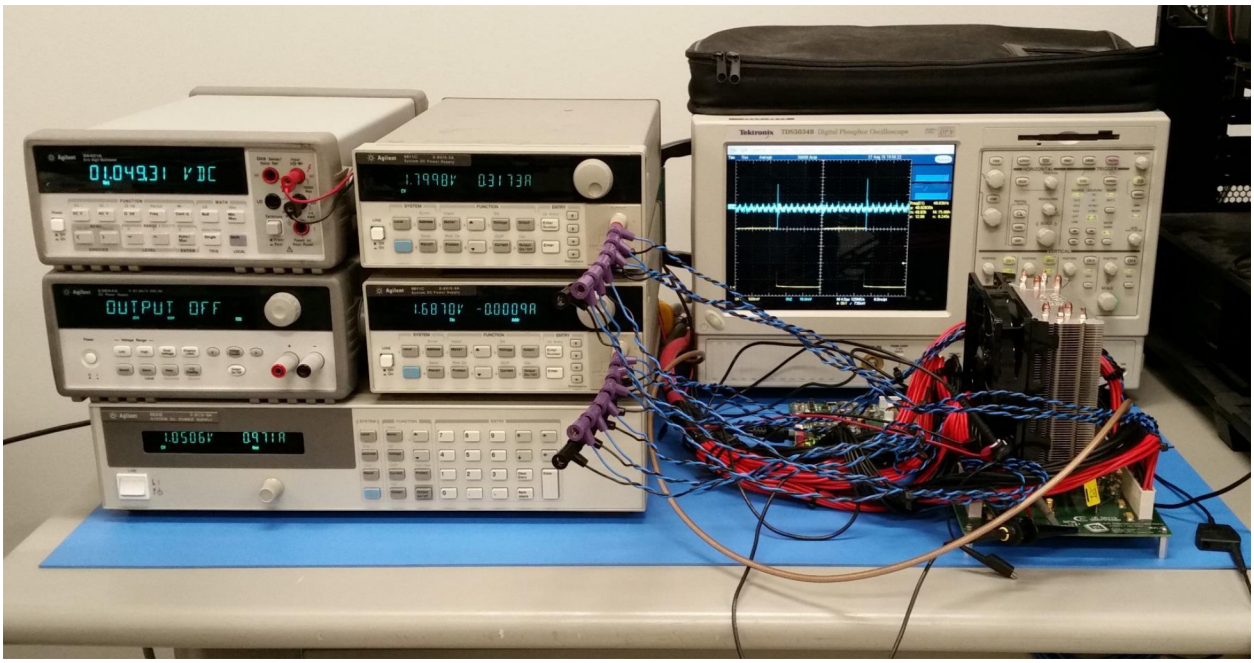


Figure 5.16: KiloCore test setup, including instruments. The Kilocore chip is mounted on a custom daughtercard which is plugged into a commercial FPGA interfacing board which is then connected to the desktop computer by a PCI Express link. Our largest optional heatsink is shown in this image.

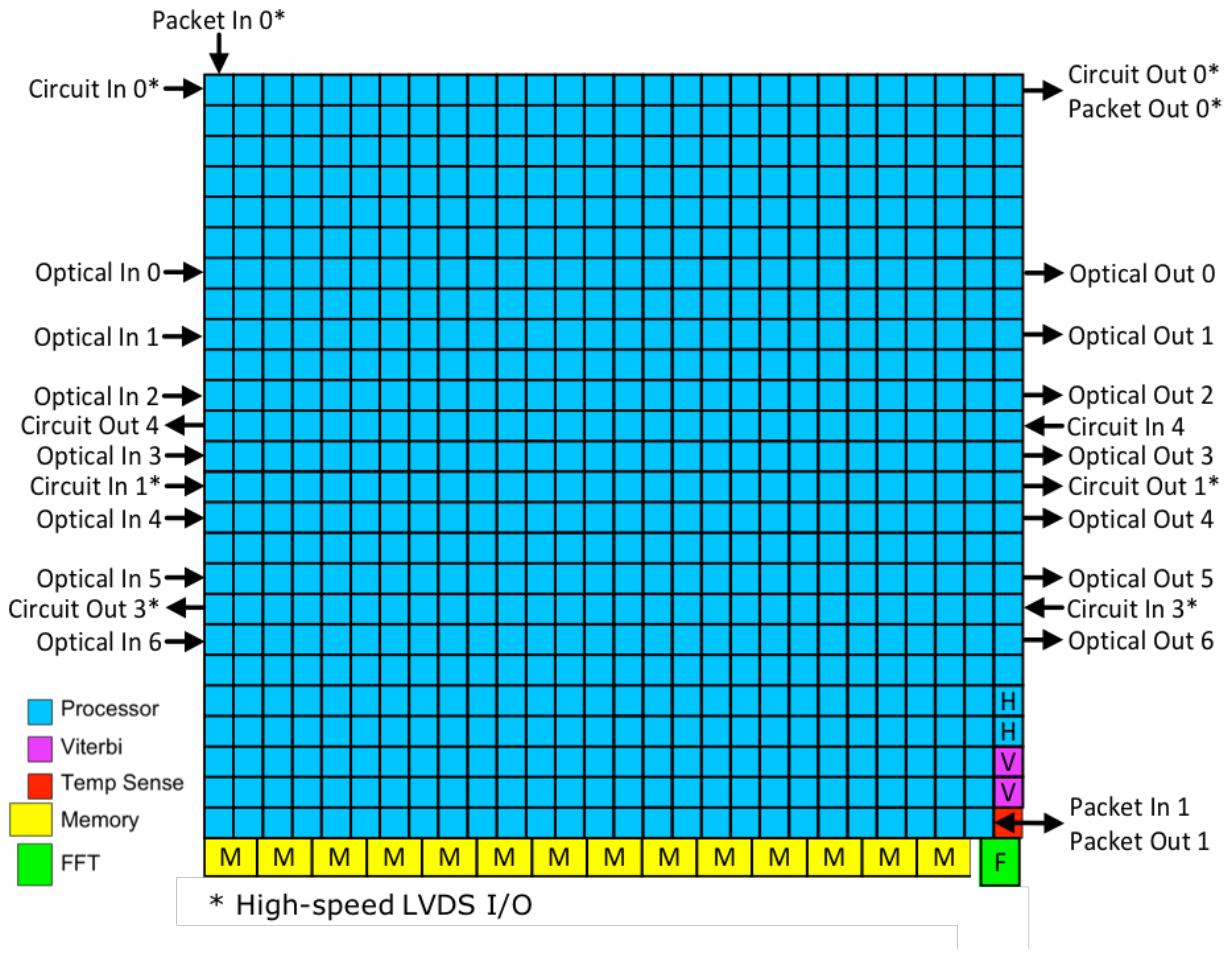


Figure 5.17: Block diagram of KiloCore2, including I/O ports.

5.2 KiloCore2

The 32 nm PD-SOI CMOS KiloCore2 integrated circuit was recently fabricated, and is awaiting a package design, to allow for testing. It contains 700 independent cores arranged in 28 rows and 25 columns. There are 695 homogeneous programmable processor cores, two high-speed programable cores, and three accelerator cores. The three accelerators consist of one FFT accelerator and two Viterbi decoder accelerators. The array also contains fourteen 65 KB independent SRAM memories. The total chip contains roughly 580 million transistors, the block diagram is shown in Figure 5.17, and a plot of which is shown in Figure 5.18. The chip contains three power rails for the cores to choose from, high, medium, and low. This allows for cores to choose a rail based on their

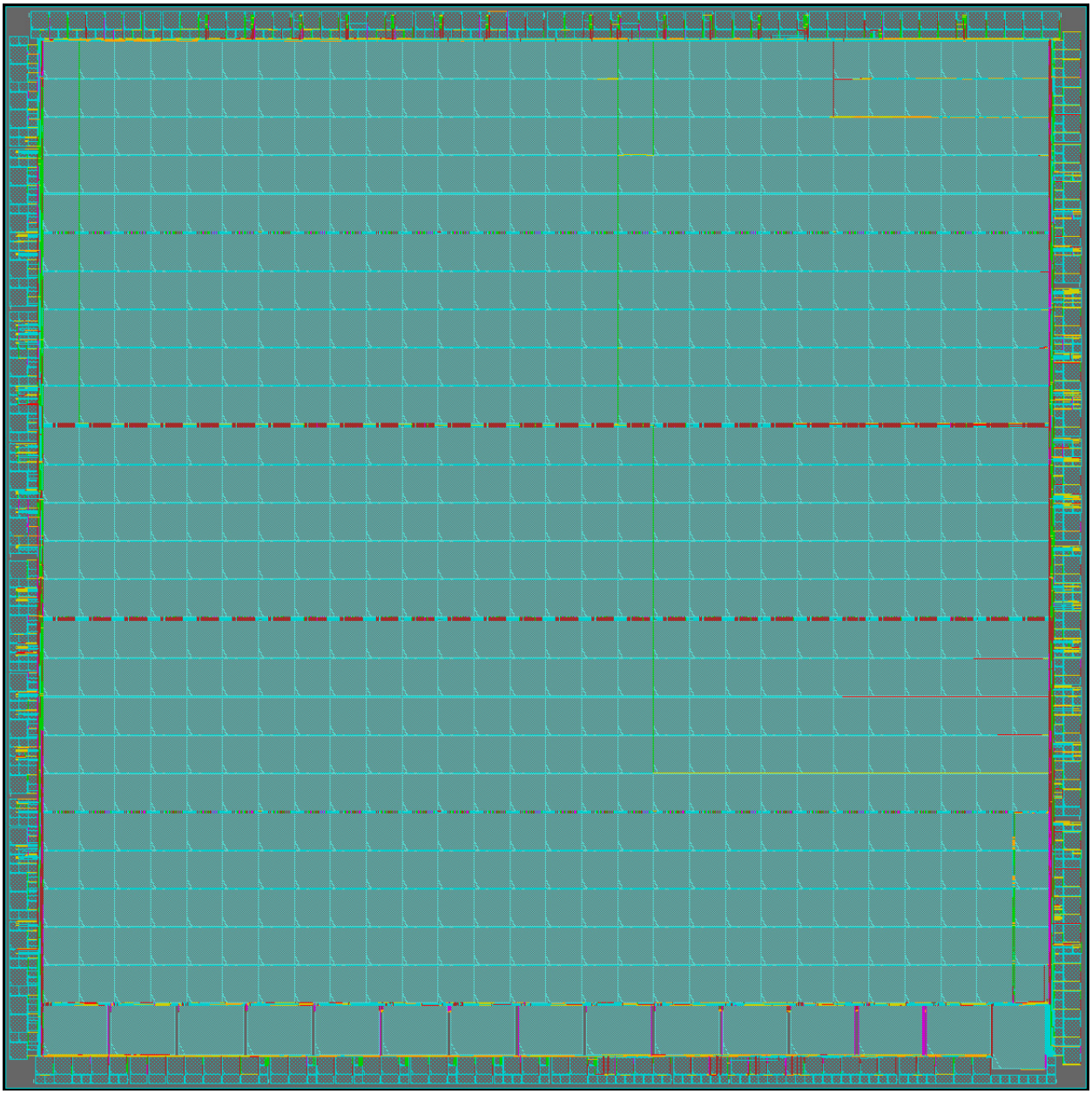


Figure 5.18: Plot from Encounter showing the placement of the array in KiloCore2 with power and ground wires hidden. The names of each of the blocks is given in the block diagram in Figure 5.17. The blocks around the periphery of the chip are I/O drivers/receivers, ESD blocks, and deep trench capacitors.

workload for energy efficiency.

The chip contains 2,499 C4 pads to connect to the package. Of the 2,499 bumps, 459 of them are I/O (149 LVDS Pairs). 2,028 are for power rails, 531 high, 200 medium, 196 low,

82 I/O, and 1,019 common ground. 12 of the bumps are analog probe points for supply for the core power domain and common ground for two separate processors and power/ground for each of the four power rails previously listed. Finally there are separate power and ground rails for the temperature/voltage sensor connected to their own C4 bumps.

5.2.1 Design

Each processor tile is 265 µm by 274.5 µm, contains 750,601 transistors, and operates with a 7 stage datapath (9 when performing a MAC operation). Figure 5.19 is a plot showing the tile, highlighting different submodules. It contains a dynamic voltage and frequency scaling (DVFS) controller which allows it to switch between desired power supply rails. Each tile contains its own independent oscillator, which allows the processor to operate at an expected 2.0 GHz. Communication between cores and chips possible via direct circuit-switched network and a completely redesigned packet router, with its own oscillator, which is expected to run at 2.0 GHz. The processors contain a 128×39-bits instruction memory, 256×16-bits data memory, two 32×16-bits FIFO communication buffers for inter-processor communication. The processor tiles is currently estimated to consume 85 mW total power with 2.3 mW leakage.

The high-speed processor, shown in Figure 5.22, was designed to perform simple tasks at high speed. This core was designed with all low-threshold, high-performance transistors, and critical paths for complicated computations were removed. The resulting design is expected to operate at around 3.85 GHz at 0.9 V. There is no DVFS in this processor, it is directly connected to the high power rail, as it is only expected to perform performance critical tasks.

There are fourteen independent memories, shown in Figure 5.23. This contains a 64 KB SRAM cell, and 3 FIFOs. This block contains 3,835,867 transistors.

There is one FFT accelerator on the chip, shown in Figure 5.24. This block performs a discrete Fourier transform, and is used in many different DSP applications.

There are two Viterbi decoder accelerator on the chip, each using 725,592 transistors, and is shown in Figure 5.25. This block is useful for decoding a data sequence that has been encoded using a "finite-state" process, and is widely used in radio communication, computer storage, and speech recognition.

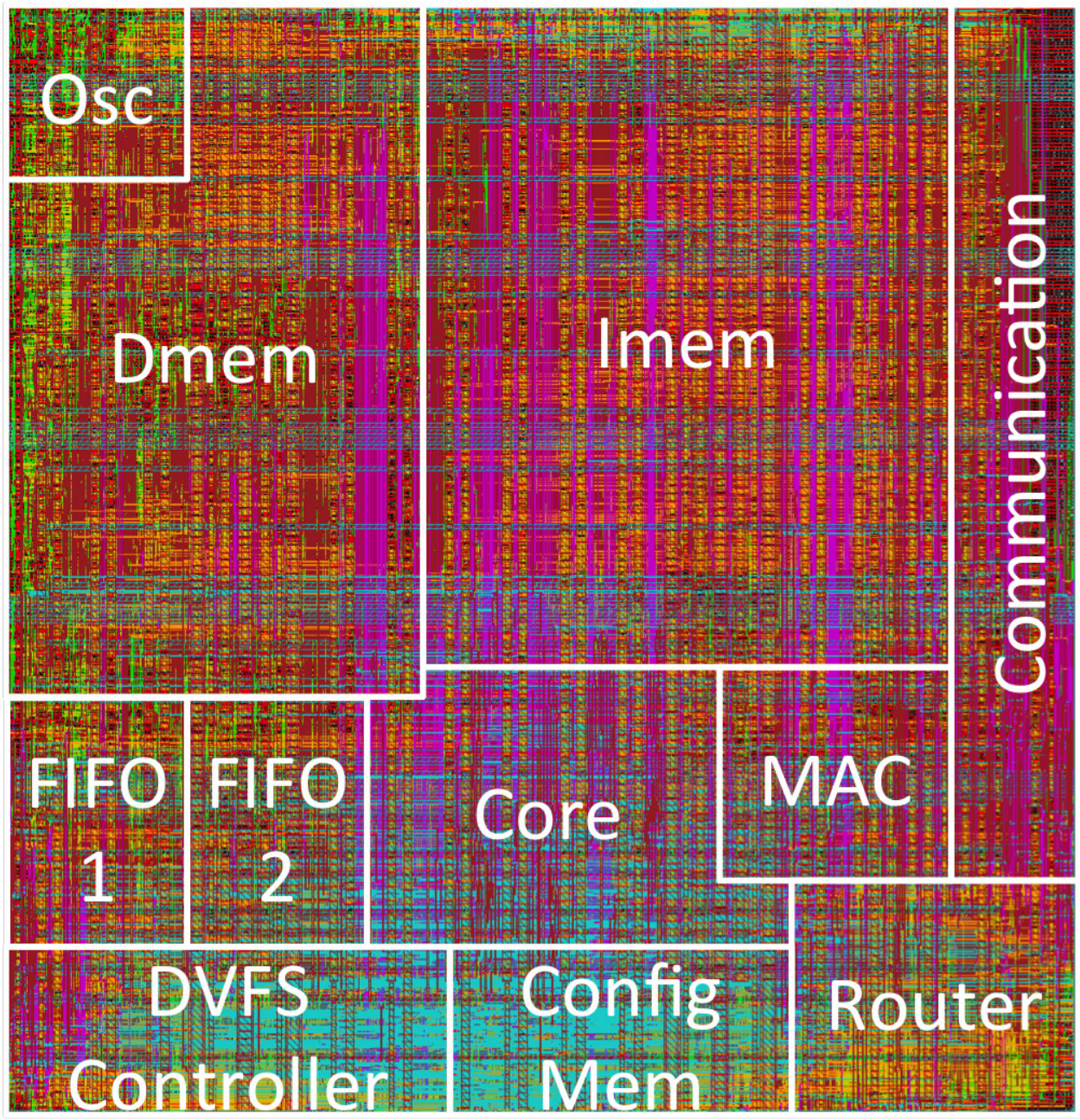


Figure 5.19: Plot of a single KiloCore2 processor, with key submodules labeled.

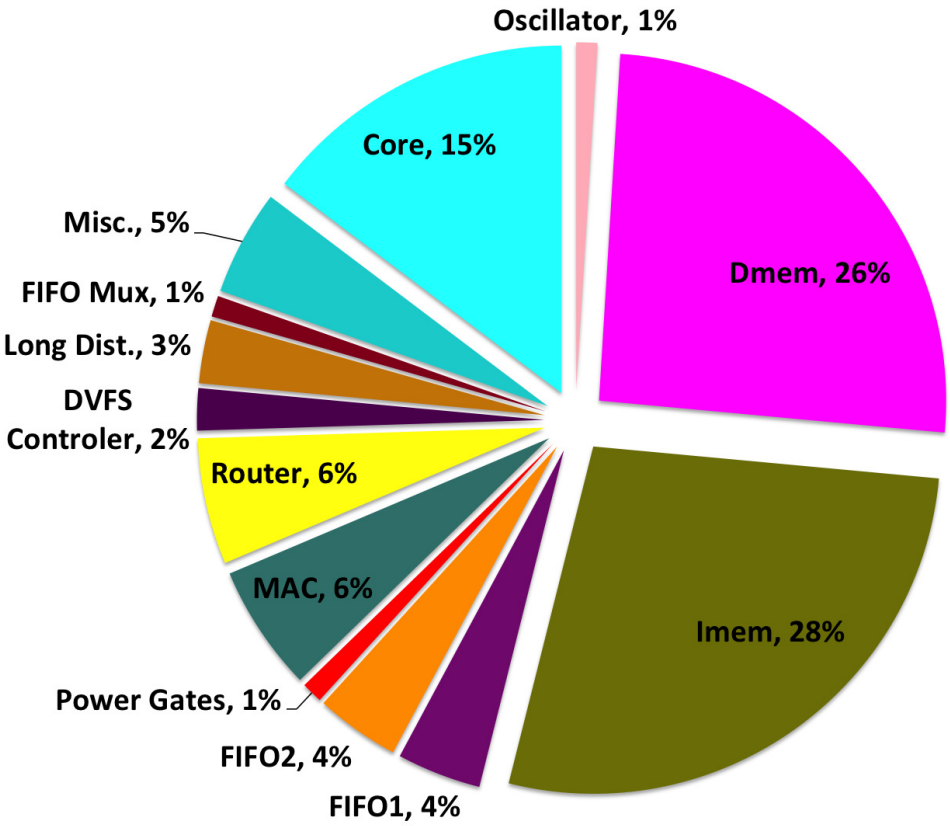


Figure 5.20: Pie graph showing the area breakdown of a KiloCore2 processor. Where these submodules are spatially located can be found in Figure 5.21.

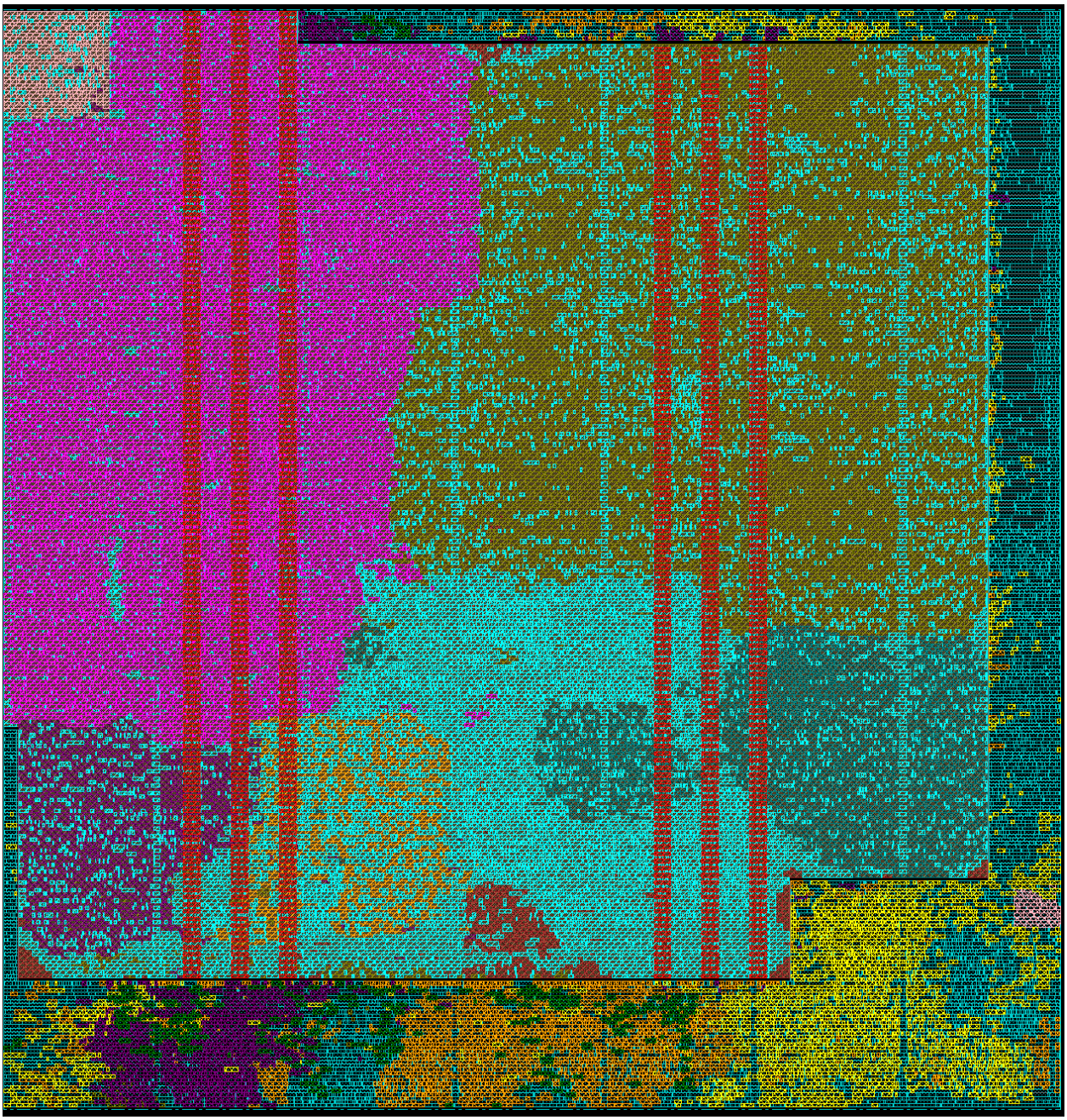


Figure 5.21: Plot from encounter highlighting standard cells used in different submodules on a KiloCore2 processor. The corresponding submodules names can be found in Figure 5.20. The always-on power domain can clearly be seen around the bottom, and the edges of the design.

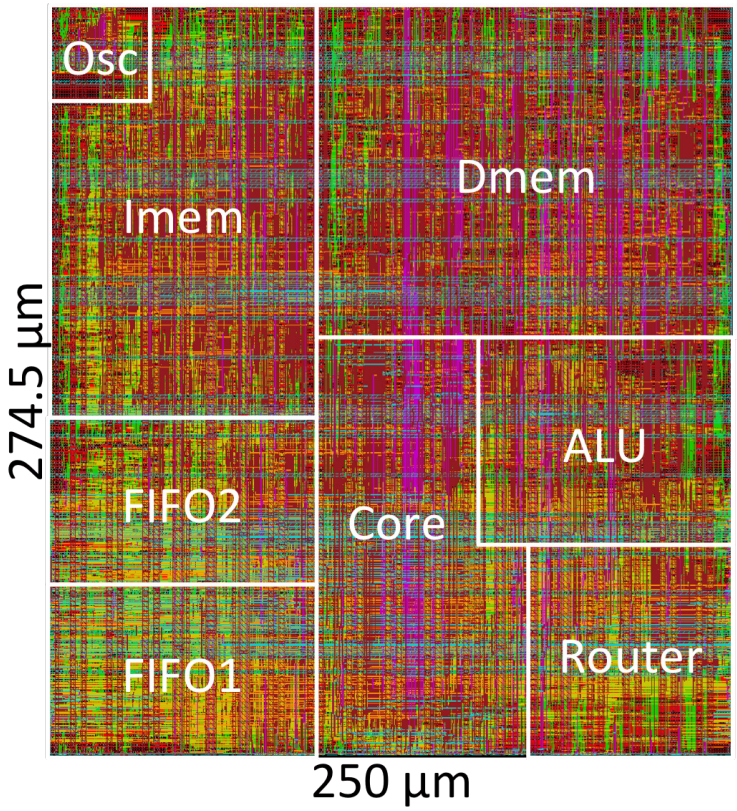


Figure 5.22: Block diagram of a single KiloCore2 high-speed processor, with key submodules labeled.

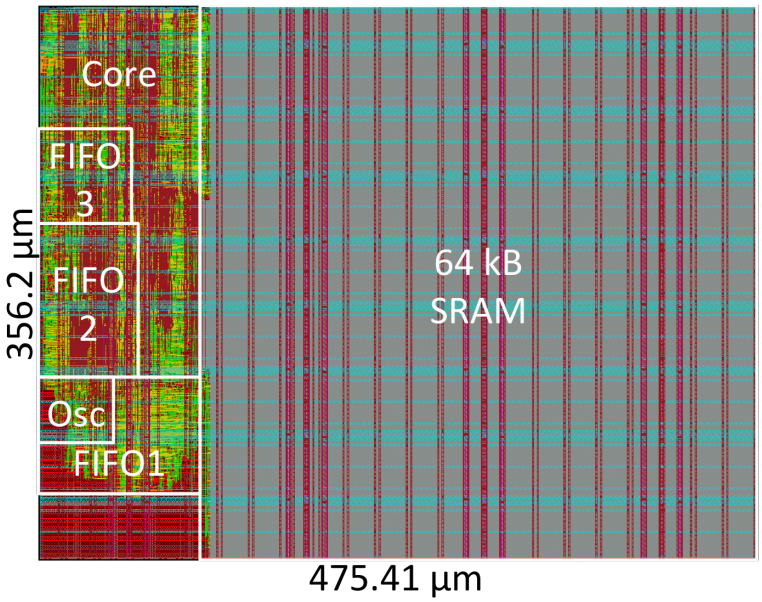


Figure 5.23: Plot of a single KiloCore2 independent memory, with key submodules labeled.

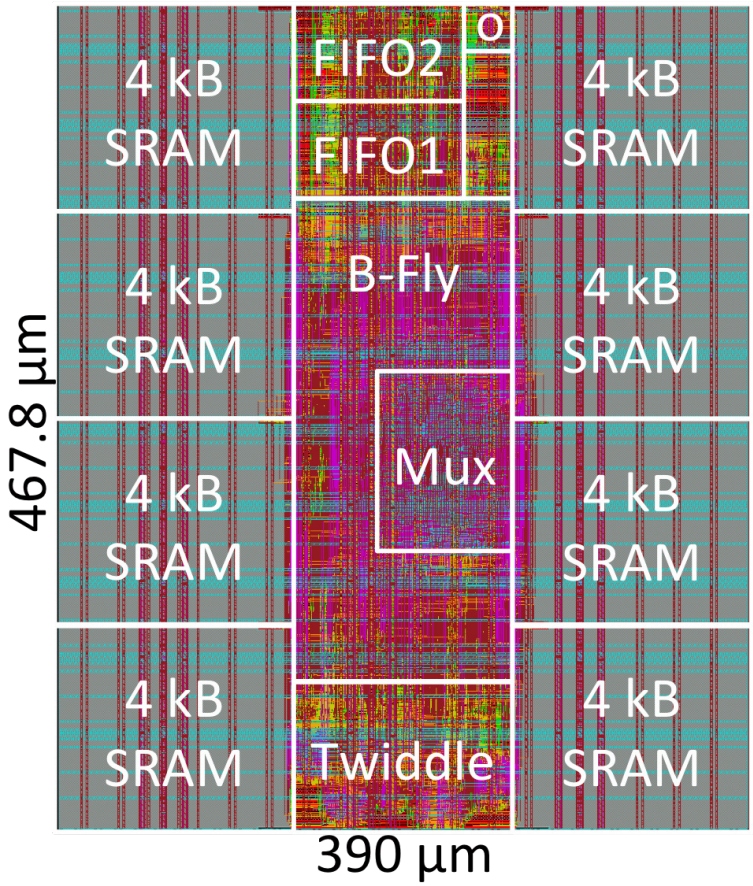


Figure 5.24: Plot of a single KiloCore2 FFT accelerator, with key submodules labeled.

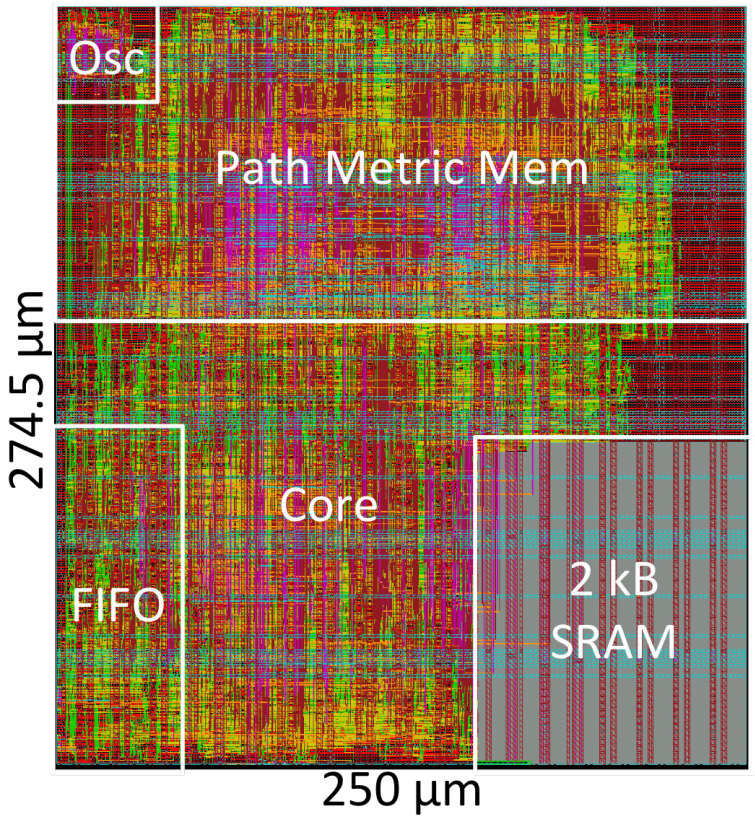


Figure 5.25: Plot of a single KiloCore2 Viterbi decoder accelerator, with key submodules labeled.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

This dissertation discusses the current research and challenges in many-core sorting algorithms, as why the topic is relevant to motivate research in many-core arrays, such as the presented KiloCore and KiloCore2 arrays. The large 2D mesh architecture used to model the presented sorting results, AsAP2, is explained. It continues to explain how the study will use different simulation methods and models, including modeling transistor scaling.

Different program kernels that together make the three many-core sorting methods that are proposed: snake, row, and adaptive. Two different methods for outputting data from the array are explained. The three sorting methods, as well as general mappings onto the chip are given. Results from a small internal sort on the many-core array are analyzed, showing that using the proposed system, the highest throughput per area is up to 27× higher and the lowest energy per record sort is more than 330× smaller than the energy per record of a similar quicksort on a general purpose CPU. The highest throughput per area is also up to 22× higher and the lowest energy per record sort is more than 750× smaller than the energy per record of a similar radix sort on a GPU.

Presented are three different sorting applications, with two different protocols for outputting data from each processor. This was accomplished on a varied number of processors in a many-core array acting as a co-processor to an Intel CPU performing the Phase 2 merge part of the external sort.

Many-core array sorts were modeled with the number of cores ranging from 10 to 10,000.

It was found that the most energy efficient phase 1 sort required over 83× less energy than the comparable Intel CPU sort and over 105× less energy than the comparable Nvidia GPU. The highest throughput many-core phase 1 sort was over 10× faster than the comparable CPU sort and over 14× faster than the comparable GPU sort. It was also shown that this same sort required over 5.5× less energy×delay than the JouleSort winner, CoolSort. The proposed sorts can be implemented on different sized arrays in a large database system and recognize large energy savings without giving up performance.

Physical design methodology for creating processor arrays in the 1000-processor era is presented. A number of issues that arise in these era are presented, such as power distribution, communication, DVFS implementation, and homogeneous module design considerations, and solutions are discussed in Chapter 4. A full physical design CAD flow is presented to handle designing large modular arrays, as current CAD tools are not specifically designed to handle all aspects of design in the 1000-processor era.

Two example many-core processor arrays, KiloCore and KiloCore2, are discussed in Chapter 5. The design and results are given for the first fabricated 1000-processor integrated circuit. At a supply voltage of 1.1 V, preliminary results show that processors operate up to an average maximum clock frequency of 1.78 GHz; therefore, one chip can execute a maximum of 1.78 trillion operations per second. At a reduced supply voltage, one chip can execute a maximum of 1 trillion operations per second while dissipating 13.1 W. At its lowest operating voltage, 0.56 V, it can perform 5.8 pJ/Op while performing 115 Billion Ops/sec. Design of the next generation, untested, chip KiloCore2, is briefly discussed, with plots given showing the different physical designs of the modules used in the array, as well as the physical design of the entire chip.

6.2 Future Work

Many-Core Sorting on Future Arrays

The sorting methods presented for the AsAP2 architecture should scale well into both KiloCore and KiloCore2, with little modification. With the increased functionality, such as the packet routers, more options are open to perform sorting on these platforms. This opens the possibility for more complex, and optimized sorts such as the odd even sort. It would be worthwhile

to see the results on the newer technology as well as explore other sorting opportunities, as it has been proven that many-core arrays offer an energy-efficient sorting platform.

Complete External Sorting Implementation

The author proposes attempting to implement an entire database sort of a data set upwards of 10 GB in a real life setup, to prove the simulated results correct. This could also allow the sorts to compete in the Sort Benchmark [88] competition. In the past I/O limits on the test setup have stopped a complete demonstration, but with recent improvements in the daughter card design [104], it may be possible for a unhindered implementation of the sort.

Further Exploration of Power Gate Utilization

Presented was an exploration of issues and solutions for the next generation of many-core processor array power gate use. However, it would be interesting to explore many more possibilities, and get quantitative results for different implementation methods, such as a ring, as well as many different dispersal patterns, and methods for chaining power gate enable signals. These tend to be long distance, highly capacitive signals, and therefore greatly affect the timing of any DVFS control logic that is managing the power gates.

Glossary

AsAP2 Asynchronous array of processors, version 2. A fine-grained many-core processor array developed at UC Davis [18].

C4 Controlled collapse chip connection. A method of connecting a chip to a package, also known as flip-chip, where solder balls are deposited on chip I/O pads to then be fused to package pads.

CAD Computer-aided design. A tool which uses a computer for design. In this context, it is used to design VLSI systems.

CMOS Complementary metal-oxide semiconductor. Technology manufactured and used in most modern digital chips.

CNT Carbon nanotube transistor. A field-effect-transistor that uses a carbon nanotube as the transistor channel.

CPU Central processing unit. A digital integrated circuit which is designed to perform general purpose computations.

Datacenter A location where many computation server and data storage systems are stored and operated.

DLL Delay-locked loop. A circuit which generates a clock signal with a related phase to a reference clock using delay elements, without the use of an oscillator.

DSP Digital signal processing. The computation or manipulation of discrete digital signals.

ESD Electrostatic discharge. A sudden event where electrical current is rapidly discharged between two potentials.

FFT Fast Fourier transform. A optimized algorithm by which to perform a discrete Fourier transform on a digital signal.

FIFO First in first out. A operation where the first value written into a buffer is the first value that will be subsequently read out.

FPGA Field-programable gate array. A digital integrated circuit which contains an array of gates which can be programmed and reprogramed to perform a certain function as desired.

GALS Globally asynchronous, locally synchronous. A computational system in which local modules maintain their own independent clock signal, so with respect to each other, the modules are asynchronous, but locally, the module's circuity is synchronous.

GPU Graphical processing unit. A specialized digital hardware unit that produces a graphical output for a computing system.

HDL Hardware description language. A computer language used to describe the functional operation of electronic hardware design.

I/O Input/output. A data port that is either used for input of data, output of data, or both.

KiloCore The first fabricated chip containing 1000 programmable, independent processing cores on a single die. It was developed at UC Davis [95].

KiloCore2 Working name of the yet untested processor array developed at UC Davis, which contains 697 programmable cores, and 3 accelerators.

LUT Look-up table. A digital integrated circuit which contains an array of constant values which are accessed based on an address.

MIMD Multiple instruction, multiple data. A type of parallel computational platform that contains multiple processing elements which each execute independent instructions.

Nanowire transistor A transistor that is created with a gate surrounding a nanowire.

NBTI Negative-bias temperature instability. A undesirable affect on digital circuits, caused by either negative gate voltages, or elevated temperature.

NEM Nano-electro-mechanical. Devices which mechanically act as switches that can be used in place of a transistor.

Netlist A description of integrated circuits and how they connect containing gate level discription, as opposed to a functional description, and may contain resistive and capacitive characteristics.

PD-SOI Partially depleted silicon on insulator. A method digital chip fabrication where a layer of partially depleted silicon, then an insulator are placed between the substrate and the transistor, so as to reduce parasitic substrate capacitance in deep submicron transistors.

PLL Phase-locked loop. A circuit which generates a clock signal with a related phase to a reference clock with the use of a local oscillator.

Processor In this dissertation a processor or core is defined to be a unit capable of independent program execution; using this definition, structures such as SIMD processing elements [14] (e.g., GPU stream processors), datapaths inside configurable arrays, or FPGA look-up table blocks, would not be considered processors.

PVT Process, voltage, temperature. PVT variations are fluctuations in integrated circuits due to process fabrication variations, variations in the supply voltage, and variations caused by temperature.

RTL Register-transfer level. A level of abstraction in digital circuit design, where the design is described in a hardware description language in terms of of combinational logic and registers.

SIMD Single instruction, multiple data. A type of parallel computational platform that contains multiple processing elements which all execute a single instruction simultaneously.

Bibliography

[1] G. E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38:114–117, 1965.

[2] M. Horowitz. Computing's energy problem (and what we can do about it). In *International Solid-State Circuits Conference*, pages 10–14, 2014.

[3] S. G. Narendra, L. C. Fujino, and K. C. Smith. Through the looking glass: the 2015 edition: trends in solid-state circuits from isscc. In *IEEE Solid-State Circuits Magazine*, volume 7, pages 14–24, 2015.

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

[5] M. Ieong, B. Doris, J. Kedzierski, K. Rim, and M. Yang. Silicon device scaling to the sub-10-nm regime. *Science*, 306(5704):2057–2060, 2004.

[6] F. Chen, H. Kam, D. Markovic, T.-J. K. Liu, V. Stojanovic, and E. Alon. Integrated circuit design with NEM relays. In *Computer-Aided Design, 2008. ICCAD 2008. IEEE/ACM International Conference on*, pages 750–757, November 2008.

[7] A. D. Franklin, M. Luisier, S.-J. Han, G. Tulevski, C. M. Breslin, L. Gignac, M. S. Lundstrom, and W. Haensch. Sub-10 nm carbon nanotube transistor. *Nano Letters*, 12(2):758–762, 2012.

[8] K. Kuhn. Considerations for ultimate CMOS scaling. *Electron Devices, IEEE Transactions on*, 59(7):1813–1828, July 2012.

[9] N. Singh, A. Agarwal, L. Bera, T. Liow, R. Yang, S. Rustagi, C. Tung, R. Kumar, G. Lo, N. Balasubramanian, and D.-L. Kwong. High-performance fully depleted silicon nanowire (diameter $\leq$ 5 nm) gate-all-around CMOS devices. *Electron Device Letters, IEEE*, 27(5):383–386, May 2006.

[10] AMD accelerates energy efficiency of APUs, details plans to deliver 25x efficiency gains by 2020. Press release, AMD, May 2014.

[11] F. Faggin, M. E. H. Jr., S. Mazor, and M. Shima. The history of the 4004. *Micro, IEEE*, 16(6):10–20, December 1996.

[12] S. Borkar. Thousand core chips: a technology perspective. In *Proceedings of the 44th annual Design Automation Conference*, DAC '07, pages 746–749, New York, NY, USA, 2007. ACM.

[13] H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam, and D. Burger. Power challenges may end the multicore era. *Commun. ACM*, 56(2):93–102, February 2013.

[14] M. J. Flynn. Very high-speed computing systems. *Proceedings of the IEEE*, 54(12):1901–1909, December 1966.

[15] J. L. Gustafson. Reevaluating Amdahl's law. *Commun. ACM*, 31(5):532–533, May 1988.

[16] M. Lehman. A survey of problems and preliminary results concerning parallel processing and parallel processors. *Proceedings of the IEEE*, 54(12):1889–1901, December 1966.

[17] P. A. Merolla, J. V. Arthur, R. Alvarez-Icaza, A. S. Cassidy, J. Sawada, F. Akopyan, B. L. Jackson, N. Imam, C. Guo, Y. Nakamura, B. Brezzo, I. Vo, S. K. Esser, R. Appuswamy, B. Taba, A. Amir, M. D. Flickner, W. P. Risk, R. Manohar, and D. S. Modha. A million spiking-neuron integrated circuit with a scalable communication network and interface. *Science*, 345(6197):668–673, 2014.

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

[19] Tesla K80 GPU accelerator. Board Specification BD-07317-001 v05, Nvidia, 2015.

[20] Z. Yu and B. 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.

[21] S. Vangal, J. Howard, G. Ruhl, S. Dighe, H. Wilson, J. Tschanz, D. Finan, A. Singh, T. Jacob, S. Jain, V. Erraguntla, C. Roberts, Y. Hoskote, N. Borkar, and S. Borkar. An 80-tile sub-100-W TeraFLOPS processor in 65-nm CMOS. *Solid-State Circuits, IEEE Journal of*, 43(1):29 –41, January 2008.

[22] S. Bell, B. Edwards, et al. TILE64 processor: A 64-core SoC with mesh interconnect. In *IEEE International Solid-State Circuits Conference*, pages 88–89, February 2008.

[23] K. Smith, A. Wang, and L. Fujino. Through the looking glass; part 2 of 2: Trend tracking for ISSCC 2013 [ISSCC trends]. *Solid-State Circuits Magazine, IEEE*, 5(2):33–43, 2013.

[24] K. J. Kuhn, M. D. Giles, D. Becher, P. Kolar, A. Kornfeld, R. Kotlyar, S. T. Ma, A. Maheshwari, and S. Mudanai. Process technology variation. *Electron Devices, IEEE Transactions on*, 58(8):2197–2208, August 2011.

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

[26] R. K. Sharma, R. Shih, C. Bash, C. Patel, P. Varghese, M. Mekanapurath, S. Velayudhan, and M. Kumar, V. On building next generation data centers: energy flow in the information technology stack. In *Proceedings of the 1st Bangalore Annual Compute Conference*, COMPUTE '08, pages 8:1–8:7, New York, NY, USA, 2008. ACM.

[27] A. Gandhi, M. Harchol-Balter, R. Das, and C. Lefurgy. Optimal power allocation in server farms. In *Proceedings of the eleventh international joint conference on Measurement and modeling of computer systems*, SIGMETRICS '09, pages 157–168, New York, NY, USA, 2009. ACM.

[28] EPA. EPA report to congress on server and data center energy efficiency. Technical report, U.S. Environmental Protection Agency, 2007.

[29] D. Niyato, S. Chaisiri, and L. B. Sung. Optimal power management for server farm to support green computing. In *Cluster Computing and the Grid, 2009. CCGRID '09. 9th IEEE/ACM International Symposium on*, pages 84 –91, May 2009.

[30] J. G. Koomey, C. Belady, M. Patterson, A. Santos, and K.-D. Lange. Assessing trends over time in performance, costs, and energy use for servers. Technical report, Lawrence Berkeley National Laboratory and Stanford University, Microsoft Corporation, Intel Corporation, and Hewlett-Packard Corporation, August 2009.

[31] G. Graefe. Query evaluation techniques for large databases. *ACM Comput. Surv.*, 25:73–169, June 1993.

[32] S. Rivoire, M. A. Shah, P. Ranganathan, and C. Kozyrakis. Joulesort: a balanced energy-efficiency benchmark. In C. Y. Chan, B. C. Ooi, and A. Zhou, editors, *SIGMOD Conference*, pages 365–376. ACM, 2007.

[33] D. E. Knuth. *The Art of Computer Porgraming*, volume 3 - Sorting and Searching. Addison-Wesley, Reading, Massachusetts, 1973.

[34] Z. Yu and B. Baas. Implementing tile-based chip multiprocessors with gals clocking styles. In *Computer Design, 2006. ICCD 2006. International Conference on*, pages 174–179, October 2006.

[35] Y. Zhang and S. Zheng. Design and analysis of a systolic sorting architecture. In *Parallel and Distributed Processing, 1995. Proceedings. Seventh IEEE Symposium on*, pages 652 –659, October 1995.

[36] E. Solomonik and L. Kale. Highly scalable parallel sorting. In *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1 –12, April 2010.

[37] D. Taniar and J. W. Rahayu. Sorting in parallel database systems. In *Proceedings of the The Fourth International Conference on High-Performance Computing in the Asia-Pacific Region-Volume 2 - Volume 2*, HPC '00, pages 830–, Washington, DC, USA, 2000. IEEE Computer Society.

[38] D. M. W. Powers. Parallelized quicksort and radixsort with optimal speedup. In *Proceedings of International Conference on Parallel Computing Technologies. Novosibirsk.*, pages 167–176. World Scientific, 1991.

[39] K. E. Batcher. Sorting networks and their applications. In *Proceedings of the April 30–May 2, 1968, spring joint computer conference*, AFIPS '68 (Spring), pages 307–314, New York, NY, USA, 1968. ACM.

[40] B. Gedik, R. R. Bordawekar, and P. S. Yu. CellSort: high performance sorting on the cell processor. In *Proceedings of the 33rd international conference on Very large data bases*, VLDB '07, pages 1286–1297. VLDB Endowment, 2007.

[41] J. Chhugani, A. D. Nguyen, V. W. Lee, W. Macy, M. Hagog, Y.-K. Chen, A. Baransi, S. Kumar, and P. Dubey. Efficient implementation of sorting on multi-core SIMD CPU architecture. *Proc. VLDB Endow.*, 1:1313–1324, August 2008.

[42] N. Govindaraju, J. Gray, R. Kumar, and D. Manocha. GPUTerasort: High performance graphics co-processor sorting for large database management. In *SIGMOD '06: Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, pages 325–336, New York, NY, USA, 2006. ACM.

[43] H. Jagadish. Sorting on an array of processors. In *Computer Design: VLSI in Computers and Processors, 1988. ICCD '88., Proceedings of the 1988 IEEE International Conference on*, pages 36–39, 1988.

[44] S. Rajasekaran. Mesh connected computers with fixed and reconfigurable buses: Packet routing and sorting. *IEEE Transactions on Computers*, 45:529–539, 1996.

[45] L. Stillmaker. SAISort: An energy efficient sorting algorithm for many-core systems. Master's thesis, University of California, Davis, CA, USA, September 2011. `http://vcl.ece.ucdavis.edu/pubs/theses/2011-2/`.

[46] A. Stillmaker, L. Stillmaker, and B. 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, December 2012.

[47] J. S. Vitter. External memory algorithms and data structures: Dealing with massive data. *ACM Comput. Surv.*, 33(2):209–271, 2001.

[48] C. Kim, J. Park, N. Satish, H. Lee, P. Dubey, and J. Chhugani. CloudRAMSort: Fast and efficient large-scale distributed ram sort on shared-nothing cluster. In *ACM SIGMOD*, pages 841–850, New York, NY, USA, 2012.

[49] N. Satish, M. Harris, and M. Garland. Designing efficient sorting algorithms for manycore GPUs. In *IPDPS '09: Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, pages 1–10, Washington, DC, USA, 2009. IEEE Computer Society.

[50] B. Baas, Z. Yu, M. Meeuwsen, O. Sattari, R. Apperson, E. Work, J. Webb, M. Lai, T. Mohsenin, D. Truong, and J. Cheung. AsAP: A fine-grained many-core platform for DSP applications. *IEEE Micro*, 27(2):34–45, March 2007.

[51] D. Truong, W. Cheng, T. Mohsenin, Z. Yu, T. Jacobson, G. Landge, M. Meeuwsen, C. Watnik, P. Mejia, A. Tran, J. Webb, E. Work, Z. Xiao, and B. Baas. A 167-processor 65 nm computational platform with per-processor dynamic supply voltage and dynamic clock frequency scaling. In *VLSI Circuits, 2008 IEEE Symposium on*, June 2008.

[52] W. H. Cheng and B. M. Baas. Dynamic voltage and frequency scaling circuits with two supply voltages. In *IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1236–1239, May 2008.

[53] Z. Yu and B. 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.

[54] A. T. Tran, D. N. Truong, and B. M. Baas. A reconfigurable source-synchronous on-chip network for GALS many-core platforms. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Trans. on*, 29(6):897–910, June 2010.

[55] S. Thompson, R. Chau, T. Ghani, K. Mistry, S. Tyagi, and M. Bohr. In search of "forever," continued transistor scaling one new material at a time. *Semiconductor Manufacturing, IEEE Transactions on*, 18(1):26–36, February 2005.

[56] J. M. Rabaey, A. P. Chandrakasan, and B. Nikolic. *Digital integrated circuits: a design perspective*. Pearson Education, Upper Saddle River, NJ, second edition, 2003.

[57] J. P. Uyemura. *Introduction to VLSI circuits and systems*. John Wiley & Sons, Inc., Hoboken, NJ, first edition, 2002.

[58] M. Bohr. The evolution of scaling from the homogeneous era to the heterogeneous era. In *Electron Devices Meeting (IEDM), 2011 IEEE International*, pages 1.1.1–1.1.6, December 2011.

[59] A. Stillmaker and B. Baas. Modeled transistor performance scaling factors of CMOS devices from 180 nm to 7 nm. *In Preparation*, In Review, November 2015.

[60] R. Dennard, V. Rideout, E. Bassous, and A. LeBlanc. Design of ion-implanted MOSFET's with very small physical dimensions. *Solid-State Circuits, IEEE Journal of*, 9(5):256–268, October 1974.

[61] G. Baccarani, M. Wordeman, and R. Dennard. Generalized scaling theory and its application to a 1/4 micrometer MOSFET design. *Electron Devices, IEEE Transactions on*, 31(4):452–462, April 1984.

[62] K. Kuhn. CMOS transistor scaling past 32nm and implications on variation. In *Advanced Semiconductor Manufacturing Conference (ASMC), 2010 IEEE/SEMI*, pages 241–246, July 2010.

[63] K. Mistry, C. Allen, C. Auth, B. Beattie, D. Bergstrom, M. Bost, M. Brazier, M. Buehler, A. Cappellani, R. Chau, C.-H. Choi, G. Ding, K. Fischer, T. Ghani, R. Grover, W. Han, D. Hanken, M. Hattendorf, J. He, J. Hicks, R. Huessner, D. Ingerly, P. Jain, R. James, L. Jong, S. Joshi, C. Kenyon, K. Kuhn, K. Lee, H. Liu, J. Maiz, B. Mcintyre, P. Moon, J. Neirynck, S. Pae, C. Parker, D. Parsons, C. Prasad, L. Pipes, M. Prince, P. Ranade, T. Reynolds, J. Sandford, L. Shifren, J. Sebastian, J. Seiple, D. Simon, S. Sivakumar, P. Smith, C. Thomas, T. Troeger, P. Vandervoorn, S. Williams, and K. Zawadzki. A 45nm logic technology with high-k+metal gate transistors, strained silicon, 9 Cu interconnect layers, 193nm dry patterning, and 100% Pb-free packaging. In *Electron Devices Meeting, 2007. IEDM 2007. IEEE International*, pages 247–250, December 2007.

[64] M. Jurczak, N. Collaert, A. Veloso, T. Hoffmann, and S. Biesemans. Review of finfet technology. In *SOI Conference, 2009 IEEE International*, pages 1–4, October 2009.

[65] S. Sinha, G. Yeric, V. Chandra, B. Cline, and Y. Cao. Exploring sub-20nm finfet design with predictive technology models. In *Proceedings of the 49th Annual Design Automation Conference*, DAC '12, pages 283–288, New York, NY, USA, 2012. ACM.

[66] A. Stillmaker, Z. Xiao, and B. Baas. Toward more accurate scaling estimates of CMOS circuits from 180 nm to 22 nm. Technical Report ECE-VCL-2011-4, VLSI Computation Lab, University of California, Davis, December 2011.

[67] Fo4 writeup: International technology roadmap for semiconductors 2003 edition. Technical report, ITRS, 2002.

[68] International technology roadmap for semiconductors. `http://www.itrs.net/`, October 2015.

[69] A. Balijepalli, S. Sinha, and Y. Cao. Compact modeling of carbon nanotube transistor for early stage process-design exploration. In *Low Power Electronics and Design (ISLPED), 2007 ACM/IEEE International Symposium on*, pages 2–7, August 2007.

[70] W. Zhao and Y. Cao. New generation of predictive technology model for sub-45nm early design exploration. *IEEE Transactions on Electron Devices*, 53(11):2816–2823, November 2006.

[71] Y. Cao, T. Sato, D. Sylvester, M. Orshansky, and C. Hu. New paradigm of predictive MOSFET and interconnect modeling for early circuit design. In *CICC*, pages 201–204, 2000.

[72] Predictive technology model. `http://ptm.asu.edu/`, October 2015.

[73] J. J. Pimentel and B. 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)*, November 2014.

[74] J. J. Pimentel, A. Stillmaker, B. Bohnenstiehl, and B. M. Baas. Area efficient backprojection computation with reduced floating-point word width for SAR image formation. In *IEEE Asilomar Conference on Signals, Systems and Computers (ACSSC)*, Pacific Grove, CA, November 2015.

[75] J. J. Pimentel, A. Stillmaker, and B. M. Baas. A SAR backprojection accelerator with reduced floating-point word width for SAR image formation. In *In Preparation*, 2015.

[76] M. Braly, A. Stillmaker, and B. Baas. A configurable H.265-compatible motion estimation accelerator architecture for realtime 4K video encoding in 65 nm CMOS. In *IEEE International Symposium on Circuits and Systems (ISCAS)*, In Review 2016.

[77] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi. Clearing the clouds: A study of emerging scale-out workloads on modern hardware. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 37–48, New York, NY, USA, 2012. ACM.

[78] D. Truong, W. Cheng, T. Mohsenin, Z. Y. T. Jacobson, G. Landge, M. Meeuwsen, C. Watnik, P. Mejia, A. Tran, J. Webb, E. Work, Z. Xiao, and B. Baas. A 167-processor computational array for highly-efficient DSP and embedded application processing. In *IEEE HotChips Symposium on High-Performance Chips (HotChips 2008)*, August 2008.

[79] B. Liu and B. M. Baas. A high-performance area-efficient AES cipher on a many-core platform. In *IEEE Asilomar Conference on Signals, Systems and Computers (ACSSC)*, November 2011.

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

[81] Z. Xiao and B. M. Baas. A 1080p H.264/AVC baseline residual encoder for a fine-grained many-core system. *Circuits and Systems for Video Technology, IEEE Transactions on*, 21(7):890–902, July 2011.

[82] Z. Xiao, S. Le, and B. 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.

[83] B. Bohnenstiehl and B. Baas. A software LDPC decoder implemented on a many-core array of programmable processors. In *IEEE Asilomar Conference on Signals, Systems and Computers (ACSSC)*, Pacific Grove, CA, November 2015.

[84] A. Stillmaker, L. Stillmaker, B. Bohnenstiehl, and B. Baas. Energy-efficient sorting on a many-core platform. In *Technology and Talent for the 21st Century (TECHCON 2013)*, September 2013.

[85] A. Stillmaker, B. Bohnenstiehl, L. Stillmaker, and B. Baas. Scalable parallel sorting on a fine-grained many-core processor array. *IEEE Transactions on Parallel and Distributed Systems*, In Review, August 2015.

[86] E. O. Adeagbo and B. M. Baas. Energy-efficient string search architectures on a fine-grained many-core platform. In *Technology and Talent for the 21st Century (TECHCON 2015))*, September 2015.

[87] D. Birru. A novel delay-locked loop based CMOS clock multiplier. *Consumer Electronics, IEEE Transactions on*, 44(4):1319–1322, November 1998.

[88] C. Nyberg, M. Shah, and N. Govindaraju. Sort benchmark home page. `http://sortbenchmark.org/`.

[89] C. Nyberg. Sort benchmark data generator and output validator. `http://www.ordinal.com/gensort.html`.

[90] C. A. R. Hoare. Algorithm 64: Quicksort. *Commun. ACM*, 4:321–, July 1961.

[91] P. Pillai, M. Kaminsky, M. A. Kozuch, and D. G. Andersen. FAWNSort: Energy-efficient sorting of 10GB, 100GB, and 1TB. Technical report, Intel Labs and Carnegie Mellon University, 2012.

[92] M. 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.

[93] R. W. Apperson. A dual-clock FIFO for the reliable transfer of high-throughput data between unrelated clock domains. Master's thesis, University of California, Davis, CA, USA, September 2004. `http://web.ece.ucdavis.edu/cerl/techreports/2004-5/`.

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

[95] 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 32nm 1000-processor MIMD array. In *In Preparation*, 2015.

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

[97] W. Huang, M. Allen-Ware, J. Carter, E. Cheng, K. Skadron, and M. Stan. Temperature-aware architecture: Lessons and opportunities. *Micro, IEEE*, 31(3):82–86, May 2011.

[98] R. Rodrigues, I. Koren, and S. Kundu. An architecture to enable life cycle testing in cmps. In *Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT), 2011 IEEE International Symposium on*, pages 341–348, October 2011.

[99] 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, October 2008.

[100] D. N. Truong and B. M. Baas. Massively parallel processor array for mid-/back-end ultrasound signal processing. In *Biomedical Circuits and Systems Conference (BioCAS), 2010 IEEE*, pages 274–277, November 2010.

[101] D. Bol, J. D. Vos, C. Hocquet, F. Botman, F. Durvaux, S. Boyd, D. Flandre, and J.-D. Legat. Sleepwalker: A 25-MHz 0.4-V sub-mm^2 7-µW/MHz microcontroller in 65-nm LP/GP CMOS for low-carbon wireless sensor nodes. *Solid-State Circuits, IEEE Journal of*, 48(1):20–32, January 2013.

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

[103] D. Markovic, V. Stojanovic, B. Nikolic, M. Horowitz, and R. Brodersen. Methods for true energy-performance optimization. *Solid-State Circuits, IEEE Journal of*, 39(8):1282–1293, August 2004.

[104] N. Mostafavi. Hardware, software, and tools for an AsAP2 many-core system. Master's thesis, University of California, Davis, CA, USA, June 2014.
`http://vcl.ece.ucdavis.edu/pubs/theses/2014-1/`.