PHOENIX: Efficient computation in memory

PHOENIX: Efficient Computation in Memory

Mats Rimborg[1]
mats@fdf.se

Pedro Trancoso[1,2]
pedro@cs.ucy.ac.cy

Gunnar Carlstedt[1]
gunnar@carlstedt.se

[1] Department of Computer Science and Engineering, Chalmers University of Technology, Gothenburg, Sweden
[2] Department of Computer Science, University of Cyprus, Nicosia, Cyprus

ABSTRACT

Parallelism is inherent in most problems but due to current programming models and architectures which have evolved from a sequential paradigm, the parallelism exploited is restricted. We believe that the most efficient parallel execution is achieved when applications are represented as graphs of operations and data, which can then be mapped for execution on a modular and scalable processing-in-memory architecture. In this paper, we present *PHOENIX*, a general-purpose architecture composed of many Processing Elements (PEs) with memory storage and efficient computational logic units interconnected with a mesh network-on-chip. A preliminary design of *PHOENIX* shows it is possible to include 10,000 PEs with a storage capacity of 0.6GByte on a 1.5cm^2 chip using 14nm technology. *PHOENIX* may achieve 6TFLOPS with a power consumption of up to 42W, which results in a peak energy efficiency of at least 143GFLOPS/W. A simple estimate shows that for a 4K FFT, *PHOENIX* achieves 117GFLOPS/W which is more than double of what is achieved by state-of-the-art systems.

CCS CONCEPTS

• **Computer systems organization → Parallel architectures → Multiple instruction, multiple data**

KEYWORDS

High-Performance Computing (HPC); Processing-In-Memory (PIM); Energy-Efficient; Parallel Architectures

1 INTRODUCTION

The maximum CPU frequency has not increased much in the last decade, and Moore's law is also approaching its end. Instead, the advent of multicore processors and parallel execution has enabled the evolution to continue. But to achieve exascale performance, a disruptive approach is needed. Simply scaling current technology will result in gigantic systems with prohibitive power requirements.

Even though most problems are inherently parallel, current programming models and architectures have evolved from a sequential model and therefore the parallelism exploited in current applications is somehow limited by this computing evolution path. We propose to disrupt this trend with a new architecture named *PHOENIX*, which stands for Parallel High-performance Optimised Energy-efficient Non-von neumann In-memory eXecution.

We have designed *PHOENIX* bottom-up from the simplest unit. We start by considering the most basic unit of the system: an *array of memory cells* of a DRAM chip. In current memory chips, these memory arrays are surrounded by logic (e.g. row and column decoding) and are connected among each other by an interconnection network used to transfer the data to and from the memory cells. In *PHOENIX*, we extend this design by surrounding the memory arrays with simple, yet versatile, compute units, forming a *Processing Element* (PE). The proposed design can deliver high-performance and energy efficiency because of the simple and general-purpose computational units that are tightly-coupled to the memory reducing considerably the wire distances required to transfer data. PEs are interconnected with an efficient mesh Network-on-Chip (NoC) used to transfer data among the PEs and in and out of the chip. The large number of PEs on the same chip allows exploiting a large degree of parallelism.

This efficient architecture is coupled with an execution model that can exploit all its potential. Applications are expressed as graphs that are partitioned into closures (subgraphs including operations and data). Closures are mapped to *PHOENIX* PEs to achieve an efficient execution. Memory accesses are efficiently hidden by the model and architecture. Accesses to local data are loaded together with the operations when a closure is selected for execution. Accesses to remote data are hidden by suspending the closure that is waiting for data and switching the execution to other closures on the same PE. A complete overview of the *PHOENIX* architecture is shown in Figure 1.

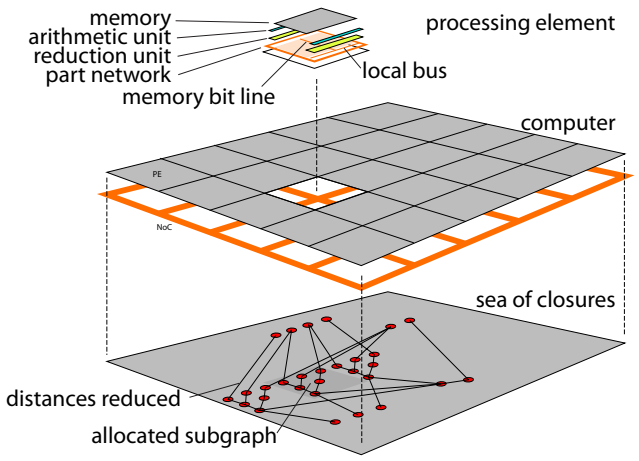

Figure 1. *PHOENIX* Architecture and graph mapping.

MEMSYS 2017, October 2–5, 2017, Alexandria, VA, USA

ACM ISBN 978-1-4503-5335-9/17/10.
https://doi.org/10.1145/3132402.3132430

In this work, we show that by augmenting small DRAM memory arrays with simple compute units, combined with an efficient execution model, it is possible to overcome the inefficiency problem in existing architectures. *PHOENIX* can achieve a high-degree of parallelism, high memory bandwidth, efficient memory latency hiding, energy efficiency, and execution of general-purpose applications.

Since *PHOENIX* can encompass tens of thousands of processing elements rather than the limited number of cores of a conventional machine, the potential for parallel execution will be significantly increased. Due to the lower power consumption, higher resource efficiency, and scalability, smaller devices will be possible to build, as well as energy efficient exascale systems. While in this work we show the potential for FFT, *PHOENIX* is a general-purpose architecture suitable for any parallel application.

A preliminary design of *PHOENIX* shows it is possible to include about 10,000 PEs with an aggregate storage capacity of 0.6GByte on a 1.5cm^2 chip using 14nm technology. At a total power consumption of up to 42W *PHOENIX* can achieve 6TFLOPS. As a case study, we show that *PHOENIX* is expected to achieve 117GFLOPS/W for a 4K Fast Fourier Transform (FFT). This is more than double the highest efficiency reported for alternative state-of-the-art architectures.

The main contributions of this work are the following. A computer architecture is proposed based on integrating small PEs around banks of embedded DRAM, all connected by a backbone NoC. Due to the proximity of compute and memory, combined with a directed acyclic graph representation of computations, mapping fine-grained computation graphs across collections of such PEs is expected to yield significant energy gains and expose the parallelism necessary to exploit a large number of PEs.

This paper is organised as follows. In Section 2 we present the related work. Section 3 covers the *PHOENIX* programming and execution model. In Section 4 the *PHOENIX* architecture is described from a logical point of view, together with a feasible VLSI implementation. This section also contains a more detailed implementation description of some essential parts. We have used the Fast Fourier Transformation (FFT) as a case study application, and the results can be found in Section 5. Section 6 is a discussion about theoretically achievable performance. Finally, Section 7 contains our conclusions.

2 RELATED WORK

Current architectures that can be used to exploit large degree of parallelism include many-core general-purpose processors (e.g. Intel Xeon Phi [1], the Cell/B.E. processor [2], Tilera/Mellanox Technologies and Cavium whose latest products TILE-Mx100™ and ThunderX2™ are both ARM-based, or Adapteva Epiphany [3], [4]). These architectures usually lack enough memory bandwidth for demanding applications (due to the von Neumann bottleneck) and their power consumption is considerably high. Even though the cores are mostly counted in hundreds rather than tens, like most commercial multicore processors they are still not sufficiently evolved. We advocate a transition to massively parallel systems, utilising thousands, and even millions of processing elements, where large problems can be distributed over a vast region and processed in an efficient manner without first splitting the problems manually. We also believe that the memory and executing parts must be tightly integrated to solve the energy and bandwidths demands that are needed in a networked society.

Another class are the accelerators aiming at executing certain parts of code efficiently, e.g. Graphics Processing Units (GPUs) (e.g. NVIDIA P100 [5]), but their power consumption is generally even higher than for conventional many-core devices. Still, given the large number of elements, GPUs typically have better energy efficiency than such devices.

Field-Programmable Gate Arrays (FPGA) (e.g. Intel Arria 10 [6]) overcome the power consumption issue, but require a significant programming effort and execution of different applications require the reconfiguration of the system.

Processing-In-Memory (PIM) architectures have been proposed for some time and the acronym PIM has been used for about 20 years. Generally, it means that there is some processor near the memory on a silicon die [7]. A key PIM project was FlexRAM which started in 1996 [8]. In 2002 Micron presented Yukon, a SIMD device with integrated logic and DRAM in a single die, intended to serve as a coprocessor [9]. It was designed as an array of processing elements, each containing an 8-bit integer arithmetic logical unit, a 128 8-bit register file, and I/O to the associated DRAM. The idea was to place many tiny cores in the memory system of an otherwise commodity machine. However, the economics of building specialised PIM systems with suboptimal DRAM integration and non-standard interfaces were unattractive to the industry. Instead, less disruptive technologies such as Multi-Chip Modules (MCM) were used.

It is only recently that such tightly integrated products have been announced, e.g. Micron's Automata Processor [10] (AP) which is claimed to be the industry's first non-von Neumann architecture and leverages the parallelism found in DRAM technology. Strictly speaking, it is more of a specialised processor built using memory rather than a PIM, since there is no application memory on an AP.

While PIM has the potential to solve the above-mentioned issues, existing architectures are either domain specific as the AP, or do not take the integration far enough. There have been numerous suggestions to bring the memory closer to the processors by using 3D-stacked DRAMs. Micron's Hybrid Memory Cube (HMC) [11] and IBM's adaptation of it [12] are examples of this. In [13] the feasibility and potential of fine-grained work migration to reduce remote data accesses in systems with multiple PIM devices is examined. In several papers emanating from AMD Research various aspects of mainly GPU-based PIM systems are investigated; in [14] hash tables for high-performance and big-data applications such as in-memory databases and key-value stores are studied, in [15] the viability of GPU-accelerated architectures as in-memory processors is explored, and in [16] the potential of using 3D die stacking to move memory-intensive computations closer to memory is examined. In [17] design choices such as caches, core frequency and number of cores for a set of Big Data analyses benchmarks based on MapReduce are evaluated for an ARM-like energy-efficient core as a PIM core. In [18] the authors show that processing-in-memory (PIM) can be a key enabler to realise memory-capacity-proportional performance in large-scale graph processing. Though all this research are steps in the right direction, the distance between the compute and memory is still too large and the bus width too narrow to enable sufficient performance increase and energy reduction. Keeping the units within a single package is in our opinion not good enough, they should be on the same chip.

There are other many-core implementations that are application specific, such as [19] where it is demonstrated how a combination of careful algorithm analysis with judiciously chosen datapath modifications allowed the authors to produce a multicore FFT architecture with excellent parallel scaling and minimal degradation in power efficiency and area efficiency of the single-core design. However, that architecture is an FFT accelerator tailored for such processing, while *PHOENIX* handles the FFT algorithm much like any other application.

To the best of our knowledge, this paper describes the first attempt to design a general-purpose processor based on a combination of near-memory computing with tens of thousands small PEs with extremely high bandwidth between compute and memory, and an execution controlled by graphs of very small processes. Thus, both the performance and the energy efficiency of *PHOENIX* is outstanding.

3 PROGRAMMING AND EXECUTION MODEL

To effectively exploit the massive parallelism in *PHOENIX*, we adopt the declarative programming model which includes dataflow, functional and logic programming languages (e.g. Erlang, Haskell and Scala). Declarative programming avoids side effects, which are commonly used in imperative programming to implement state and I/O. This provides for referential transparency, i.e. expressions can be replaced by their values without changing the behaviour of the program. This makes it easier to verify, optimise, and parallelise programs, and facilitates designing automated tools to perform those tasks. Even though this can be applied to programs written in any language, declarative programs are easily mapped into graphs of operations and data.

The *PHOENIX* architecture does not use any cache memory as the computation elements are integrated into the memory, resulting in very small access latency. Without caches, it is possible to have a simpler system as there is no need for coherency support. The memory accesses are performed as atomic transactions, which ensure the synchronisation between processes. This makes the concurrent memory semantics extremely simple.

3.1 Graph Representation

The *PHOENIX* architecture differs from the von Neumann model in that it does not contain any global program counter, and the program is not stored as an instruction sequence. Instead, the processes which together constitute the program are represented as directed acyclic graphs (DAG), where the nodes are rewritten to their simplest (canonical) form. The graph symbolises the relationship between the inputs to the program, and its outputs. Each node in the graph is an expression. Each node generally has dependencies to other nodes, represented by the arcs of the graph, and these dependencies control the execution. Expressions can be of various kinds: atomic values, identifiers or references to memory, tuples of expressions, or functions. The simplest kind is an atomic value (e.g. an integer or a real number). Other inputs include references to other subgraphs, which corresponds to the memory addresses in conventional systems. An expression can also be a tuple, which may contain an arbitrary number of expressions. Conditional statements are executed by discarding subgraphs that do not fulfil the condition. Finally, functions are expressions that can be applied to one or more arguments (e.g. arithmetic operators). A graph for a simple expression is shown in Figure 2. In addition to static graphs like the one in the example, the model and architecture can also support the dynamic creation of subgraphs.

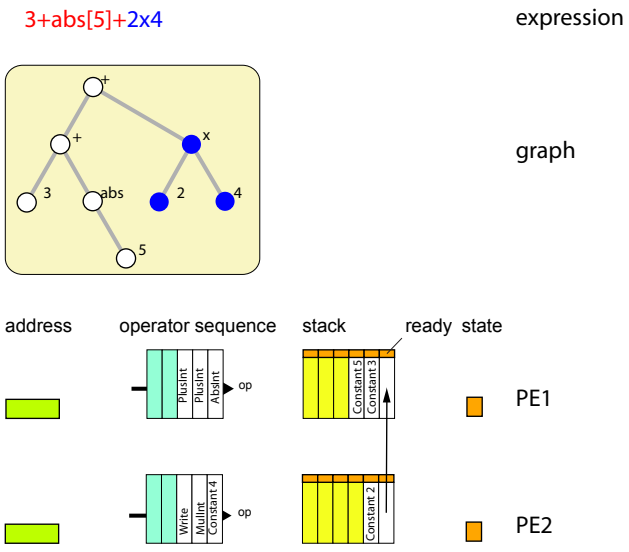

Figure 2. A simple expression on top, its graph representation in the middle and its execution as two closures on two different PEs.

3.2 Mapping

The computer system may be viewed as a large surface of memory cells, see Figure 3. On this surface, the program graphs are placed. This graph is rather independent of the language in which it was described, but primarily dependent on the application. The set of data, references, and operations is known as a *closure*. Each closure can be considered as a process, or a part of a process. The closures control parallelism and synchronisation. Notice that the graph in Figure 2 has been partitioned into two closures represented with nodes in different colour (light and dark nodes).

To increase parallelism closures should be distributed over many PEs, however, the communication network is loaded proportional to the aggregate length of the arcs of the graph. Proper allocation will reduce this length. This is a shortest-path minimisation problem for the arcs of the application graph. Closures in separate PEs execute in parallel, and the allocation does not affect the result of their evaluation. Furthermore, each PE can store multiple closures. Closures are mapped to the same PE to hide the latency of memory accesses to remote PEs. The PE structure is so simple and the access to local memory so fast that the switching between closures is done in two 2ns memory cycles (one write and one read). This increases the potential for parallel execution significantly.

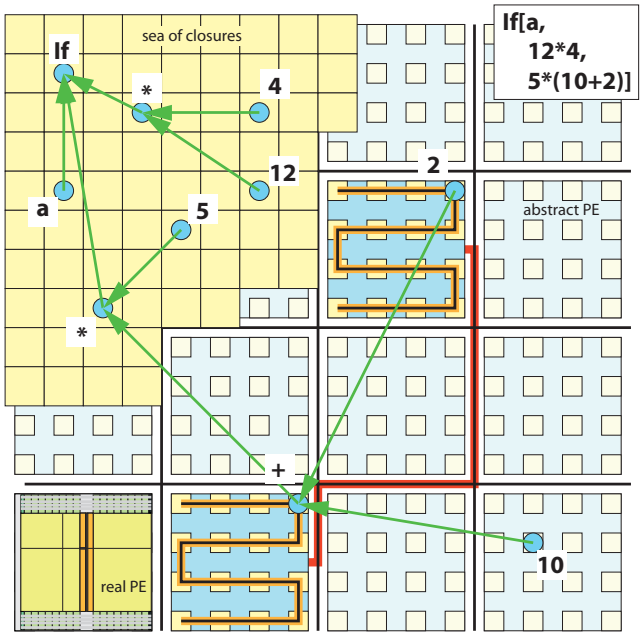

Figure 3. Memory region.

3.3 Execution

The execution mechanism aims to evaluate the expressions. This is achieved by *reducing* the graphs. An execution mechanism which is suited for fully distributed systems controls the allocation and scheduling of the problem. This control mechanism is used to decide which nodes in the graph perform rewrite operations. It can balance the sometimes contradictory requirements between latency time and available processing resources.

The scheduling of closures is made in parallel. The scheduling of the graph inside a closure is made so that the implementation becomes as simple as possible, and the accesses are thus sequential. The graph is therefore rewritten into a tree, where shared subgraphs are located closer to the root. The first of the outermost leaf nodes is scheduled and is followed by the others so that an operator gets all its operands, see Figure 4.

Scheduling is done by traversing from the outermost first subtree and continuing with the next larger tree until the whole tree has been traversed with pre-traversal scheduling. This order can be considered as a list of pointers to the nodes that are examined.

On the bottom of Figure 2 the two closures of the expression and graph in the same Figure execute on two different PEs (PE1 and PE2). All "dark closure" operations are executed in PE2 without interruption until the end, which is a *Write* operation. In PE1 the "light closure" operations are also executed without interruption and in parallel with PE2 until the second operator *PlusInt*, which requires a stack element that is not ready (meaning that is being produced somewhere else). At that point PE1 performs a closure switch bringing in another one which is ready for execution. The data produced by PE2 is written to the memory address of the closure's stack position and the ready bit is set. Next time the "light closure" is selected for execution it will continue until the end.

As has been shown above, an execution can be specified by a list of operators and a stack. All execution within the subgraph can occur independently and take place sequentially. It is therefore important to use efficient algorithms that allow as much parallelism as possible. When the subgraph is not itself a leaf, it refers to other subgraphs. These use the lower entries of the stack. Because of the parallel execution, these may at any time be in either state *ready* or *not ready*, indicated for each stack entry, see Figure 2. When attempting to access a *not ready* entry, the execution will stop and wait until it gets *ready*. Hence the subgraph also has an execution state; *waiting*, *pending*, *evaluating* and *canonical* (i.e. finished).

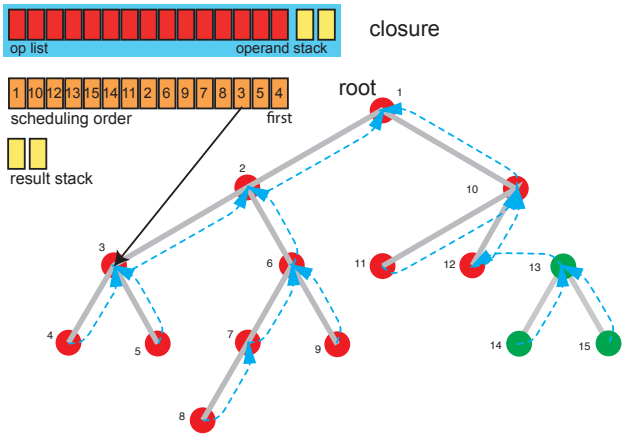

Figure 4. Depth-first pre-order traversal.

4 PHOENIX ARCHITECTURE

This chapter includes a detailed description of the structure of a Processing Element (PE). It is made up of memory, reduction unit and auxiliaries. The latter consist of oscillator, voltage regulator, etc., but are ignored here. The purpose is to illustrate simplicity, size, energy consumption and latency time. The statistics of the FFT algorithm, see Section 5, have been used to calculate the energy consumption.

First, the elements of the PE are described. This is followed by a short description of the interconnection network. The third part contains technology descriptions and shows a feasible VLSI implementation of a *PHOENIX* chip. The remainder of the chapter describes implementation strategy and details which are not needed to understand the overall functionality of the *PHOENIX* architecture. However, it contains useful information for those who wish to go deeper into the design.

4.1 Processing Element

The PE (Figure 5) consists of a *Reduction Unit* and *Memory*. The silicon area used for the Reduction Unit is just a fraction of the memory, with a ratio of about 1:4, while the Reduction Unit can store a single closure and the memory contain up to 512 closures. The Reduction Unit is a *stack machine* that executes the closure operations. This enables the storage to be used very efficiently.

In this paper, the word width is assumed to be 32 bits which is adequate for many signal processing applications. However, the *PHOENIX* architecture does not impose any restrictions on this, so depending on the characteristics of the application domain other design choices can be made.

There is an *address* register which selects a closure-wide (≈1,100 bits) set of words in the memory. The tight integration between the Reduction Unit and the Memory provides a great benefit for performance, effectively removing the von Neumann bottleneck. A part of the closure, *oplist*, contains the operators of the closure (64 8-bit operators), and another part, the operand *stack*, contains the operands (16 32-bit words). The buses *a* and *b* hold the operands from the *stack*, while *op* holds the operator from *oplist* during execution. The units *alu*, *normalise* and *mask* perform integer arithmetic, adjust exponent and mantissa in real numbers, and perform logic shifts. The bus *bits* is used for communication to and from the PE. The *control unit* decodes *operator* to perform control of the other units.

The key to performance is to have one closure stored in the high-speed registers of the Reduction Unit and using a simple arithmetic unit. At a point in time, the reduction unit contains one closure (in execution). Closures are swapped between the memory

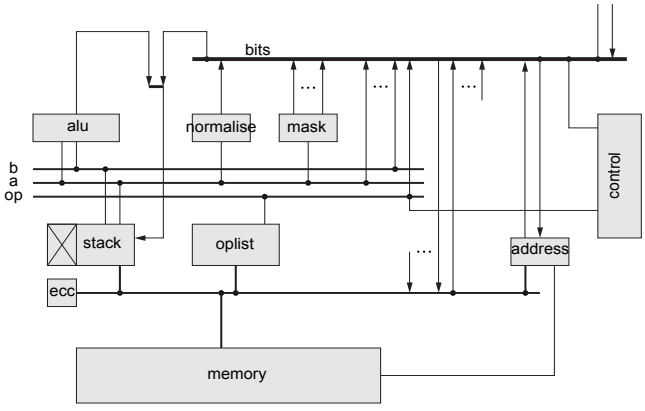

Figure 5. Block diagram of a *PHOENIX* PE.

and the reduction unit, with two 2ns memory cycles (one write and one read) using the bandwidth much more efficiently than in conventional machines.

All memory is seen by the reduction units as a single address memory space. There are no caches in the system and thus no need for complex coherency support. The memory latency is not an issue for the execution as operations to local data are performed from the stack. Local data is part of the closure and is therefore loaded into the reduction unit's stack when the closure is selected for execution. Latency to remote memory accesses is hidden by switching the execution to other pending closures in the same PE.

4.2 Interconnection Network

To reach operands located in the memory of another PE a message exchange takes place over the network. This includes an identifier of the source closure and the address of the memory destination. In the case of a read access, the receiving PE sends back a message with the content. This sets a *ready* flag in the source closure which makes the closure pending, enabling it to be swapped back into the Reduction Unit for further processing. In case of a write access, a confirmation is sent back to ensure synchronisation.

The interconnection network consists of word-wide *links* and *nodes* connected as a mesh. The node has five duplex links: one to a PE and one to each N, E, S, and W direction. Multiple *PHOENIX* chips can be merged to large systems, using the same communication protocol between chips as within a single chip.

4.3 PHOENIX Chip

4.3.1 Structure. A feasible VLSI implementation has been devised, which consists of a grid shaped, package switched network. Within each rectangle of the grid, there is a PE with a 64KByte memory block (layout shown in Figure 6). Using 14nm technology, each element has an area of just 0.015mm^2.

The small size enables a clock frequency of 6.67GHz, with a power dissipation of maximum 4.2mW including the network. Integer and real 32-bit performance will be in the range of 6.67GOPS and 600MFLOPS respectively. A 1.5cm^2 chip contains about 10,000 PEs. The chip area is divided into 10% network, 18% for the reduction unit and 72% memory. The peak power consumption for this chip is 21W for the operations, and 21W for the network. Due to the scalable design, various chip sizes can be produced for different needs.

The memory industry has been reluctant to PIMs because the reduction unit requires a more complex technology. By surrounding the memory with simple processing units, and using some few custom designed bit slices, it will be possible to reduce the number of layers significantly.

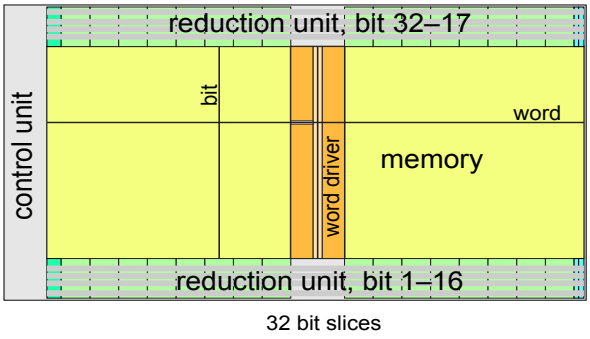

Figure 6. *PHOENIX* PE layout.

4.3.2 Technology. We have used relevant published work as base for our design assumptions, chip area, power consumption and latency estimations. The PE is described by a lumped RC network. Power is estimated by switching capacitance. The delay is calculated by longest path of RC-delays. However, all values presented are preliminary and no detailed simulations have yet been performed.

The memory design is based on [20]. In [21] it is shown how large memories are built from small blocks.

The VLSI wires are based on the lower metal levels of [22] describing a 9 metal 22nm layer technology and [21] describing an advanced 22nm 15-layer version. A subsequent 14nm version is shown in [23]. The reduction of wires, both in length and numbers, is a main reason for the ability to increase clock frequency while at the same time reducing the power consumption, compared to other multiprocessor chips.

Regarding the transistors used, we base our work on the characteristics defined in [24]. For the power and propagation delay calculation we used the current value of 730µA/µm and 210nm for the gate width.

We use eDRAM technology for the memory block as is already being used for the large on-die L3 cache in the IBM Power8 [25]. We assume the access to memory to be 2ns [26].

4.3.3 Circuitry. The PE consists of a memory block with memory cells and address drivers, and a reduction unit. The memory part uses dynamic memory technology. For the remainder of the chip, a technology based on FinFETs and CMOS is used.

The lowest conductor layer is used to contact transistors. Two subsequent narrow-pitch metal layers are used for local wires within cells. In the next layer, a low-capacitance technology is used for long wires, such as networks, controls and buses. In the two following layers, thicker conductors are used for voltage distribution; GND and VDD. In total this adds up to six layers. Additionally, there are bit and word-wires within the memory.

Each register consists of two cascaded inverters with a transmission gate that closes the loop. A few registers are of master slave type and have a subsequent register connected via a transmission gate. The registers can directly feed other devices. Output can be done bi-directionally via a transmission gate to a bus. Input can be done separately via several parallel transmission gates.

Drivers consist of inverters with several parallel transistors (fins). High drive capacity is obtained by cascading multiple drivers with an exponentially increasing drive capacity and linearly increasing latency.

Bus drivers are made in a complementary manner: a wire can be set to 0 and 1 or to a certain input value. It is implemented by a tristate driver consisting of a complementary pair of transistors driven by two cascaded gates.

The PE consists of just a few types of parameterised cells. The use of random logic is negligible.

4.4 Implementation Strategy

As was described earlier (Figure 3), the arcs of a graph can be implemented as accesses from one PE to another. When a transport follows a route in a network to another PE and back, it passes links formed by wires. Each link has a capacitance that is switched. An arc therefore has an aggregated capacitance proportional to the total length of the route links. The complete graph has an aggregated capacitance of all its routes. The energy consumption is proportional to the capacitance, hence proportional to the total route length. Therefore, the *allocation of a graph should be as dense as possible.*

When a route is divided into links, the capacitance of a link is less than for the route, but the aggregated capacitance of all the links is constant. Therefore, the energy consumption is independent of how the route is segmented into links. Each link is clocked at a frequency f (Hz) and has a length l (metres). The transport speed is the product f × l. The actual transport speed can be lower due to congestion when several arcs are active. A valid measure for transport capacity in a parallel system shall reflect the number of parallel routes. For a square grid, the distance between links will be equal to the link length l and in the general case, with a rectangular grid, proportional to l. The number of parallel routes will therefore be proportional to 1 / l, and the transport capacity proportional to (f × l) × (1 / l) = f.

In a purely sequential execution, the propagation delay (access time) for an arc is proportional to the route length. Hence, the latency time for the entire graph is proportional to the aggregated length of the route links. This also implies that the *allocation of a graph should be as dense as possible.*

If we on the contrary assume a completely parallel execution with an infinite capacity of the network, the latency time of a graph corresponds to the aggregated propagation delays along the longest path from the root to a leaf. Therefore, the *allocation of a graph should form the shortest routes from root to leaves.*

A certain graph has an access pattern which causes a latency time. This is inversely proportional to the clock frequency f. Therefore, a *high clock frequency is preferred.*

A high-performance network can be defined as links driven by transistors with a fixed resistance. The propagation delay of a link will be proportional to its capacitance, thus its length l. The clock frequency is inversely proportional to the propagation delay. In a lightly loaded system with no congestion the access time for a route is independent of the link lengths, but the transport speed will be inversely proportional to the link lengths. Therefore, *short links are preferable.*

All accesses involve the memory. Assume that the memory has a certain capacity; depth × width. The form factor of the memory can be changed by adjusting its width. Regardless of this, one word wire and all bit wires are used for an access. The total length of the bit wires is proportional to the capacity but independent of the width. Each memory cell has a fixed capacitance of a bit wire. Low capacity of a memory block is therefore preferred. The access time is proportional to the length of the bit wires, and hence the bandwidth inversely proportional to the bit wire lengths. *Therefore, the memory should be wide.*

Operands are allocated on top of a stack. Since the execution is hierarchical, operands belonging to an operator are always located at the top of the stack. By replacing the pointers in the scheduling order with the corresponding operator, the graph can be eliminated and replaced by a list of operators, see Figure 4. The closure therefore consists of a complete state for executing a subgraph; state, operand list and operand stack. One such is directly mapped in the hardware as a reduction unit. *By making the closure small, wire lengths can be limited, thus reducing energy consumption and propagation delays.*

The size of the operand stack has been determined from earlier experience, however, the *PHOENIX* architecture does not impose any restrictions on this, so depending on the characteristics of the application domain other design choices can be made. Eight words are usually too few and 32 words are unnecessarily many. This has *determined the operand list length to 64 bytes and the size of the operand stack to 16 words.*

By implementing the two upper words in the operand stack as fixed registers, the propagation delay from a clock pulse is minimal; that of an inverter. Arithmetic operations always use the same registers; therefore, no additional multiplexers are required. The energy consumption is therefore also minimal.

Similarly, the operator list is optimal. A complete 64-byte wide operator list is read from a register. Via transmission gates and tristate drivers, the operator is put on a bus whose length approximately corresponds to the width of the memory. This bus cannot be made any shorter. As a result, energy consumption and propagation delay from clock can hardly be lower.

The arithmetic can be divided into word / Boolean / numerical arithmetic, carry and permutation of bits in words. By gathering all logic for the word / Boolean / numerical arithmetic within a bit slice, the length of the wires can be kept very short, <10 µm. By performing permutation of bits on a word wide bus (32 bits), the total switched length can be limited to the word length times the width of the memory. If logic circuits or multiplexers were used this would require a significantly greater length. The carry chain is of great importance. The length of this corresponds to the memory width and can hardly be made much shorter. With appropriate implementation, its size, energy consumption and propagation delay can be minimised.

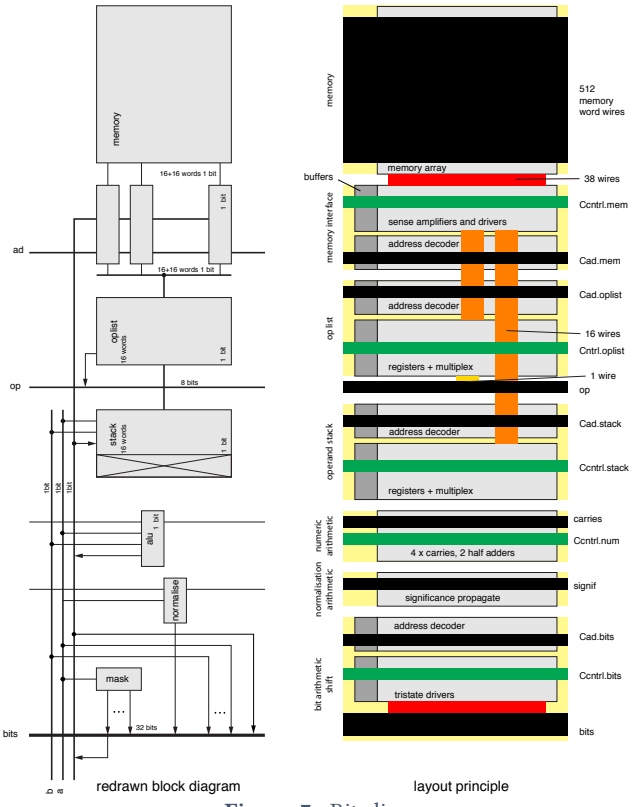

Figure 7. Bit slice.

4.5 Implementation Details

4.5.1 Memory. A memory access switches all bit wires and one word wire. The total energy consumption for this is 2.7 pJ, of which 99.6% originate from the bit wires. The proportion of the entire PE's consumption is 47%. Reading and writing of a word or a full closure takes place in the same way and thus has the same energy consumption.

4.5.2 Bit Slice. Along the edge of the memory there are *bit slices*. The parts of these are described from the one closest to the memory and outwards.

The reduction unit consists of 32 bit slices. Each such implements one and the same bit in a 32-bit word. The memory is connected with bit wires to the bit slice. The width of the bit slice interfacing the memory corresponds to the height of the operand stack; 16 words, and the 16 words in the operator list, as well as ECC bits. Typically, this makes up 16 + 16 + 6 = 38 bits. The width of the bit-slice is 10.2 µm, see Figure 7.

4.5.2.1 Memory Interface. Closest to the memory there are interfaces for the 38 bit wires located next to each other. These are classical free swinging cross coupled inverters used as sense amplifiers. They can be set either from the memory or from the rest of the bit slice. To the bit slice, 16 + 16 bits can be set (by reading and writing) in parallel via a bus. A single flip-flop can also be read or written. For this, a 5-bit address decoder is used in each interface. The properties of the interface correspond to what is commonly used in dynamic memories.

4.5.2.2 Operator List. The operator list consists of 16 parallel 1-bit registers, see Figure 8. These can be written from either operator or operand stack part in the memory through two transmission gates. For each register, there is a decoder 0–15 of the operator address CadOps. These control transmission gates that read one bit from the 16 registers. An additional 2 bits of CadOps are used by a decoder to select byte in 32-bit words. A tristate driver drives the operator bus Op.

The control unit has a register PC, see Figure 8. This is strongly buffered to drive the CadOps wires. These are internally buffered in each bit slice and feed the decoders. These open a transmission gate that feeds the local bus. The addressed byte drives the bus Op, which is then fed into the register Op.

In a sequential execution, there is on average one wire switching of the aggregate CadOps wires per operator.

The local bus is switched at every fourth operator. The eight Op wires are switched up to four times per operator. Thus, it is mainly Op that determines the energy consumption. The total energy consumption is 0.139 pJ. Of these, 0.109 pJ (78%) originate from the bus Op. The portion of the entire PE's consumption is 6.7%.

The delay is minimal, see Figure 8.

4.5.2.3 Operand Stack. The operand stack consists of 16 parallel registers, see Figure 9. These feed the Memory Interface via a bus and thus transportations can take place in either direction. A transmission gate is used to interface the bus. The values of the operand stack are stored in the register 0 and up. The two top words are always stored in registers 14 and 15, and these are named B and A. They feed two slave registers that are part of the arithmetic unit. The other consist of a register and two transmission gates that feed a local memory bus for reading and writing. Each register has a decoder that uses four bits from CadStack. Via control signals, these feed the two transmission gates. Writing is done in register CadStack + 1 while reading is from register CadStack.

The control unit has a register TOS, see Figure 9. This is strongly buffered to drive the CadStack wires. These are internally buffered within each bit slice and feed the decoders. The Write stack and Push micro operations write a register from the arithmetic unit. In the Pop and Dyadic micro operations, a value is output from a register. Via the arithmetic unit, this is forwarded to the B or A register. The arithmetic unit is fed from the two slave registers A and B.

Typically, TOS is incremented or decremented by one for each cycle. By using Gray code only one bit is switched for each step. In the loops for multiplication and division, TOS is constant and only the registers A and B are written. Otherwise, two or three registers are written for each operator. With a given memory technology, it's hard to achieve less energy consumption and shorter access time to the operands. The total energy consumption is 0.118 pJ for a dyadic operation. The control signals and the CadStack address uses 0.107 pJ (85%) of this. The portion of the entire PE's consumption is 19%.

4.5.2.4 Numerical and Logical Arithmetic. Arithmetic takes place between the A and B registers. Below the operator stack there is a device that resembles the classical 74181 integrated circuit. The unit consists of logic that forms approximately two half-adders. These can be controlled so the device performs logical operations or addition / subtraction. To reduce the latency, the first half-adder is controlled directly from two bits of the operator register OP.

The Carry chain consists of three levels, see Figure 10. The top level is a carry that flows between the 16 lower and 16 upper bit slices on each side of the memory. Within each such group of 16 bit slices, a carry is used which flows between four groups of bit slices. Within each such group, two carry chains are used which calculate if the sum ≥16 and ≥15. For the final calculation within

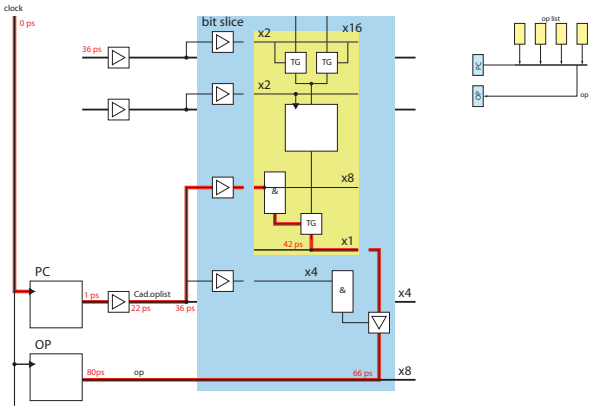

Figure 8. Operator list.

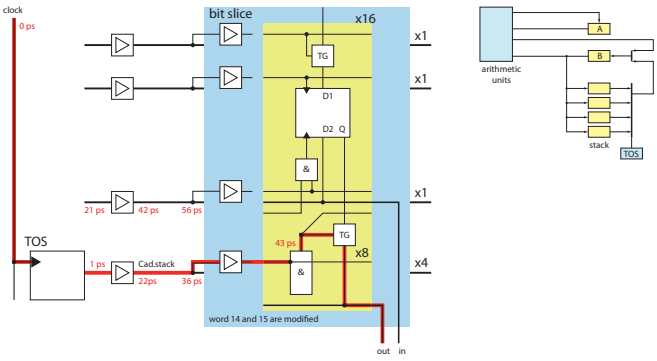

Figure 9. Operand stack.

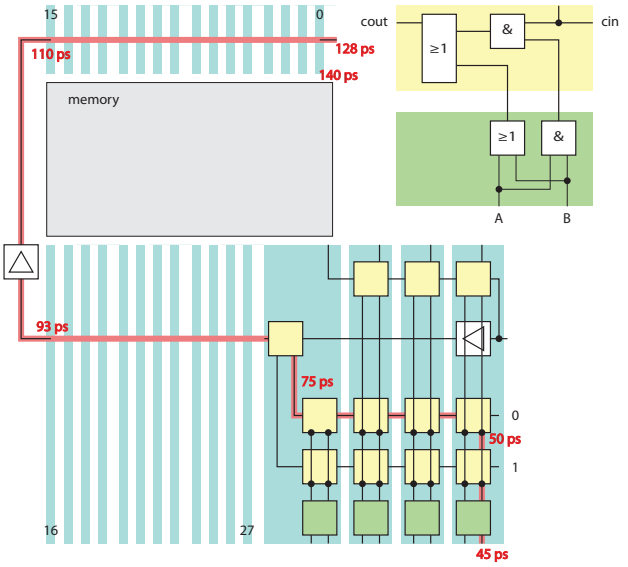

Figure 10. Carry chain.

the group, an additional carry chain is used that calculates the actual input carry value.

The power consumption is determined by the carry chain. This consumes 0.077 pJ. Of these, 0.063 pJ (82%) are lost in the wires. The proportion of the entire PE's consumption is 13%. The use of a look-ahead carry would require even more wires, thus increasing the surface and power consumption.

4.5.2.5 Normalisation Arithmetic. The normalisation arithmetic calculates the number of shift steps to be performed to normalise a result. The unit consists of an exclusive or for each bit that converts the result so that it corresponds to a negative number. A ripple chain of AND gates indicates the propagation of the most significant bit. A gate above such a step indicates where the change occurs. This bit controls an operation in the unit Bits. It outputs to the bus bits the number of steps to shift.

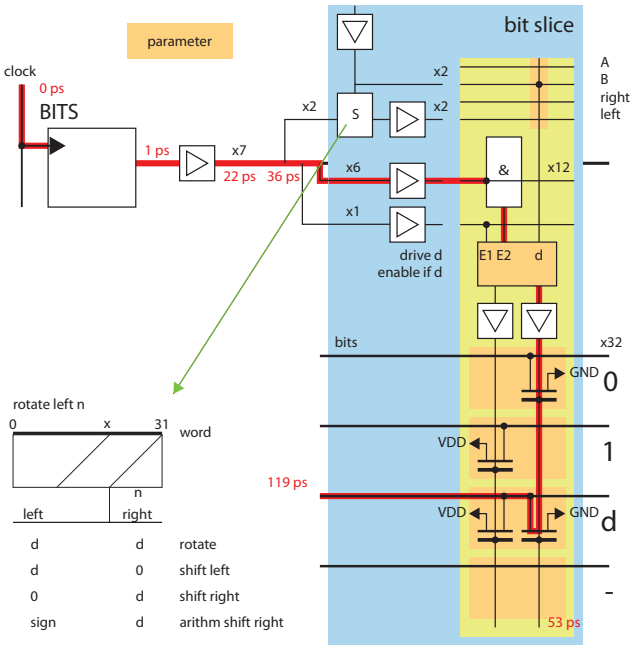

Figure 11. Bits arithmetic.

The implementation is analogous to the ripple carry discussed earlier, but simpler. The energy consumption is negligible.

4.5.2.6 Bit Arithmetic and Shift. This unit outputs a value to the bus bits. The value can be a calculated constant, a certain value or a shift of an operand. Generally, the operation involves most bits in an operand. The unit consists of tristate drives that can drive an individual wire on the bus bits to 0 or 1, see Figure 11. The unit is controlled by the bus CadBits. In each bit slice, there are many drivers. These are addressed with CadBits. Selected drivers output a value on the bus bits. The value is obtained by a simple logic that selects 1, 0 or the value of an operand. The driver can also work as in a ROM to output a value to parts of or to the entire bus. About 40 drivers are required.

For shift operations, the output of the driver is configured so that CadBits specifies a rotation to the left relative to the bit slice. Three values can be selected by CadBits for output: a) the operand or sign bit; b) the operand or 0; or c) 0. This enables all shift and rotation operations. For this, 32 drivers are required.

Permutation of operands A and B are used to generate a floating-point number or to decode such a number. Outputs can be done by registers A or B. A normalisation shift count can be made.

In the control unit, there is a BITS register, see Figure 11. This is strongly buffered to drive the CadBits wires. The energy consumption for a typical operation is 0.921 pJ. Of these, CadBits use 0.132 pJ (14%) and the bus bits 0.436 pJ (47%). The proportion of the entire PE's consumption is 24%.

4.5.3 Control. The control consists of one unit for each of PC, TOS, OP and multiplication / division. All these units share a similar structure. The device functions like a PLA. As an example, the control from OP is used. This device uses the OP register in conjunction with some internal bits to specify micro operations for an operator as well as the state of a closure.

The unit has a result bus R and an input bus I, see Figure 12. In one plane, AND operations are performed based on I. All bits need not be used. For each AND operation there is a driver that may set the bits R individually to 1 or 0. The implementation uses two rows of transistors. In the AND plane one row of transistors is connected in series while the other is connected in parallel. The

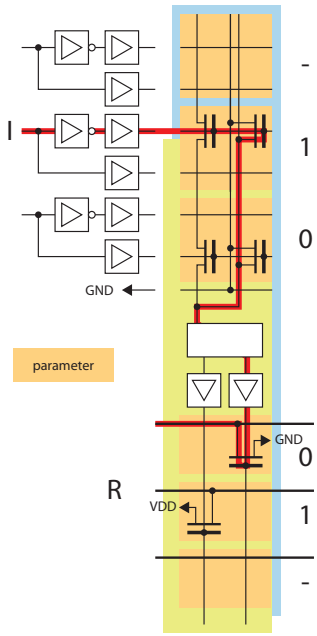

Figure 12. Programmable logic array.

output is buffered and output to two complementary wires in the R plane. One wire can use a transistor to drive a wire in R to 0 and the other 1. The use of these transistors in I and R buses is parameterised. The bus R can also be connected to another device such as the one for multiplication.

5 EVALUATION

5.1 Problem: FFT

For this study, we have used the Fast Fourier Transformation (FFT) as a case study application. This is one of the most stressful algorithms for computer architectures, with a lot of arithmetic operations and low locality. FFT is used in many technical applications (e.g. MP3 and JPEG compression). However, *PHOENIX* is a general-purpose architecture suitable for any parallel application.

By deploying a parallel software FFT algorithm on a many-core processor, a significant speed-up may be obtained compared to sequential execution. However, due to the properties of the commonly used Cooley-Tukey algorithm [27], it is challenging to handle the massive reordering of input data during the evaluation. In addition to processing power, the FFT implementations also require fast continuous exchange of data between processing nodes.

5.2 Estimated Execution and Energy Results

Figure 13 depicts an 8-input FFT, handling 8 samples at a time. The diagram contains 12 2-input butterflies. The Wx,y values are called twiddle factors, and are constants. Where arrows meet, a complex addition or subtraction is performed, as shown in the corresponding node.

For an FFT size of 4,096 words (4K FFT), an estimation of *PHOENIX* performance has been made. The implementation is systolic, i.e. there is a chain of processes where each one performs a sequence of operations on data that flows between them. It consists of 12 stages with 2,048 butterflies in each stage. Each butterfly is a process stored in one closure, with two complex input parameters stored as four words in the stack. The result is passed on by storing it in other butterflies' stacks. The store operations control the synchronisation; as soon as both inputs are ready the closure's status is set to *pending* and thus it may be loaded into the reduction unit for execution.

The systolic chain is divided into 4 parts, A, B, C, and D, with 3 stages of 4,096 complex inputs in each. The parts are placed within a rectangular area of 64 × 32 or 2,048 PEs distributed among the different parts as shown in Figure 14. In this distribution, all needed communication is between neighbouring areas of PEs.

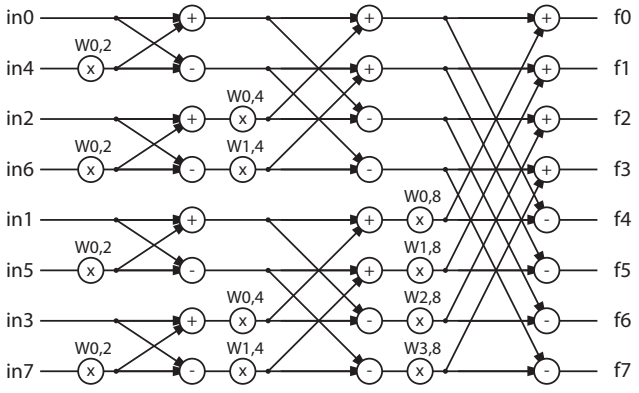

Figure 13. 8-input FFT butterfly.

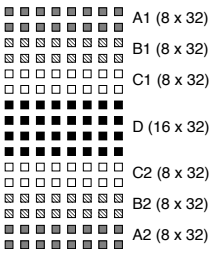

Figure 14. PE assignment for 4K-FFT.

Each PE contains 12 butterflies in separate closures. Within each PE the closures are executed concurrently, by switching from closure to closure whenever an operation needs to wait. The administrative process for scheduling the closures is very simple. The operators execute in 4 cycles of 0.15ns (cycle time at 6.67GHz) for each administered butterfly. There are 2ns for internal memory accesses resulting in a total of 2.6ns for closure administration.

The operations for a 2-input butterfly consist of 4 multiply, 5 addition and 1 subtraction of real arithmetic operators, as well as the load and store operations. In total the execution time is 31.3ns. This means that for each PE, which has 12 closures (12 butterflies), the total execution is 407ns. The maximum communication latency between the different areas (A to D) is 51ns. This determines the performance of the systolic chain. Together with the PE execution time it gives a total of 458ns.

An estimation of the energy consumption for this operation, considering the processing elements and the network used, results in approximately 4.6W.

The performance achieved for the 4K FFT is then 536GFLOPS, and the energy efficiency 117GFLOPS/W. In Figure 15 we compare the energy efficiency for different state-of-the-art architectures. We use the results reported by the manufacturers in [1], [3], [4], [5], [6], [28], [29] for the peak efficiency and for the 4K FFT implementations, if available. From the results, we can see that *PHOENIX*'s energy efficiency is more than double as compared with other state-of-the-art architectures for both peak and 4K FFT implementations.

6 DISCUSSION

For a specific algorithm, an efficient allocation can be made of its DAG in memories. There is a limit for the execution performance of this DAG in terms of surface, power and speed. Above, it has been shown that power consumption and latency time are mainly determined by the long wires in the memory, the control of bit slices, communication with one word between bit slices and carry chains, as well as networks. These are fundamental and cannot be avoided. By changing the layout, its lengths can be affected. Below, optimal performance is discussed for a certain memory

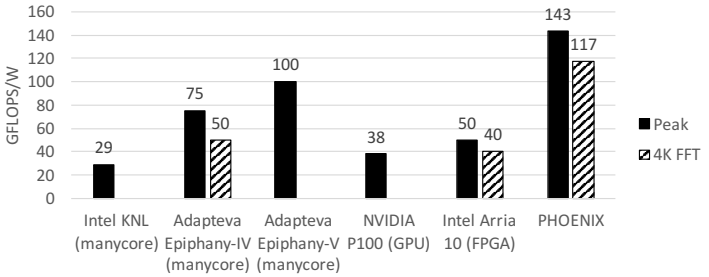

Figure 15. Peak and 4K FFT energy efficiency for different architectures.

width. On average, each clock cycle of the PE uses 0.36 pJ, an operator consumes 2.08 pJ and a FLOP consumes 7.71 pJ.

The general mechanism of a processor is reading instructions, decoding and providing control. The operation code has an average length of just over eight bits. From the PC register, access, decoding and generation of about 30 control signals are made. The energy consumption for one operator is 0.578 pJ and is mainly proportional to the width of the memory. The energy for the operator list can be considered minimal for the memory. It is unlikely that an entirely optimal implementation would have significantly fewer control signals. Overhead for control signal generation is 316% in terms of energy consumption.

All programmed logic requires at least two registers for dyadic operations. More registers increase bandwidth but also energy consumption. A dyadic operation excluding control consumes 0.017 pJ, compared to just 2 registers 0.0051 pJ, with an overhead of 233%.

The simplest arithmetic unit is a full adder with ripple carry chain. Excluding wires this consumes 0.015 pJ. The numerical arithmetic unit uses 0.072 pJ, with an overhead of 221%.

Even if these overheads could be eliminated, the main energy consumption from the memory will remain 2.8 pJ per access. By making the memory smaller, energy consumption would decrease and the chip surface increase. By using different storage order in the closures and the rest of the memory, the memory can be divided into parallel banks. At a width of 128 + 16 = 144 bits, the address decoder overhead is relatively small. The eight parallel banks can then be accessed in parallel when a closure is accessed, and only one of them when a word is accessed. Such an access would consume 0.370 pJ or 7.4 times lower than shown architecture. This should be appropriate but has not been investigated. For FFT, this design change would reduce the energy consumption of the PE by not more than 23%.

Consider that a theoretically optimal device has no surface, consumes negligible energy and has insignificant latency compared to what has been described as minimal. Individual units for the PE are several times worse than theoretically best device. By changing the form factor of the memory, minor improvements can be made. With the use of a special process such that the pitch for a bit slice would fit the pitch of the memory bit wires, wire lengths could decrease 2.5 times. Together with the use of memory banks, this could reduce the energy consumption 2–3 times.

By making the memory four times smaller and changing the form factor, energy consumption could be reduced about three times for the PE. However, the surface becomes roughly double and the network's energy consumption will therefore increase by approximately two times.

In this range, i.e. about 0.6 pJ per operator there is a threshold. Conventional processors would consume more power. *PHOENIX* is not far from this threshold.

7 CONCLUSIONS

A new type of processor-in-memory architecture for general-purpose execution has been shown. Raw performance for *PHOENIX* with 10,000 PEs in 14nm technology is estimated to 6TFLOPS, with 1.5cm^2 chip size and 42W power consumption. The parallelism and efficiency are well beyond any current architecture.

Our estimated results show that for a 4K FFT, *PHOENIX* achieves 536GFLOPS on 2,048 PEs resulting in an energy efficiency of 117GFLOPS/W, which is more than double the reported alternative state-of-the-art 4K FFT implementations. Encouraged by these preliminary results, we are currently implementing a simulator of the *PHOENIX* architecture.

ACKNOWLEDGMENTS

The authors wish to thank Professor Per Stenström at Chalmers University of Technology for his involvement and valuable contributions to this paper.

REFERENCES

[1] A. Sodani et al., "Knights landing: Second-generation intel xeon phi product," *IEEE Micro*, vol. 36, no. 2, pp. 34–46, Mar 2016.

[2] T. Chen, R. Raghavan , J. N. Dale , E. Iwata, "Cell broadband engine architecture and its first implementation: a performance view," *IBM Journal of Research and Development*, v.51 n.5, p.559-572, September 2007.

[3] A. Olofsson, "Epiphany-V: A 1024 processor 64-bit RISC system-on-chip," *CoRR*, vol. abs/1610.01832, 2016. [Online]. Available: http://arxiv.org/abs/1610.01832

[4] A. Olofsson, "How to build a Megacore microprocessor," *HiPEAC17*, 2017. [Online]. Available: https://www.parallella.org/wp-content/uploads/2017/01/million_cores.pdf

[5] NVIDIA, "Nvidia tesla p100," NVIDIA Corporation, Tech. Rep. WP-08019-001 v01.1, 2016.

[6] A. Vishwanath, "Enabling high-performance floating-point designs: Unleash high-performance floating-point processing capabilities with arria 10 fpgas and socs," Intel Corp., Tech. Rep., 2016.

[7] G. Loh et al, "A processing in memory taxonomy and a case for studying fixed-function PIM," *46th IEEE/ACM International Symposium on microarchitecture (micro-46)*, 2013.

[8] J. Torrellas, "FlexRAM: Toward an Advanced Intelligent Memory System – A Retrospective Paper," *IEEE 30th international conference on computer design (ICCD)*, 2012.

[9] G. Kirsch, "Active memory: Micron's Yukon," *Proceedings International Parallel and Distributed Processing Symposium (IPDPS)*, 2003.

[10] P. Dlugosch et al., "An efficient and scalable semiconductor architecture for parallel automata processing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 12, pp. 3088–3098, 2014.

[11] J. T. Pawlowski, "Hybrid memory cube (HMC): Breakthrough DRAM performance with a fundamentally re-architected DRAM subsystem," in *Hot Chips 23*. Boise, ID, USA: Micron Technology, Inc., 2011. [Online]. Available: http://hotchips.org/wp-content/uploads/hc_archives/hc23/HC23.18.3-memory-FPGA/HC23.18.320-HybridCube-Pawlowski-Micron.pdf

[12] R. Nair et al., "Active Memory Cube: A processing-in-memory architecture for exascale systems," *IBM Journal of Research and Development*, vol. 59, issue 2/3, pp. 17:1–17:14, 2015.

[13] Paula Aguilera, Dong Ping Zhang, Nam Sung Kim, Nuwan Jayasena, "Fine-Grained Task Migration for Graph Algorithms Using Processing in Memory," *IPDPS Workshops 2016*, pp 489-498

[14] Alex D. Breslow, Dong Ping Zhang, Joseph L. Greathouse, Nuwan Jayasena, Dean M. Tullsen: Horton Tables, "Fast Hash Tables for In-Memory Data-Intensive Computing," *USENIX Annual Technical Conference 2016*, pp 281-294

[15] Dong Ping Zhang, Nuwan Jayasena, Alexander Lukashevsky, Joseph L. Greathouse, Lifan Xu, Michael Ignatowski. "TOP-PIM: throughput-oriented programmable processing in memory," *HPDC 2014*, pp 85-98

[16] Dong Ping Zhang, Nuwan Jayasena, Alexander Lyashevsky, Joseph L. Greathouse, Mitesh R. Meswani, Mark Nutter, Mike Ignatowski, "A new perspective on processing-in-memory architecture design," *MSPC@PLDI 2013*, pp 7:1-7:3

[17] Marko Scrbak, Mahzabeen Islam, Krishna M. Kavi, Mike Ignatowski, Nuwan Jayasena, "Processing-in-Memory: Exploring the Design Space," *ARCS 2015*, pp 43-54

[18] J. Ahn, S. Hong, S. Yoo, O. Mutlu, K. Choi, "A scalable processing-in-memory accelerator for parallel graph processing, ", *ISCA 2015*.

[19] A. Pedram, J. McCaplin, A. Gerstlauer, "A highly efficient multicore floatingpoint FFT architecture based on hybrid Linear Algebra/FFT cores," *JSPS*

[20] S. Natarajan et al., "A 14nm logic technology featuring 2nd generation finfet, air-gapped interconnects, self-aligned double patterning and a 0.0588μm^2 sram cell size," in *2014 IEEE International Electron Devices Meeting*, Dec 2014, pp. 3.7.1–3.7.3.

[21] S. Narasimha et al., "22nm High-Performance SOI Technology Featuring Dual-Embedded Stressors, Epi-Plate High-K Deep-Trench Embedded DRAM and Self-Aligned Via 15LM BEOL," in *2012 International Electron Devices Meeting*, Dec 2012, pp. 3.3.1–3.3.4.

[22] D. Ingerly et al., "Low-k iterconnect stack with metal-insulatormetal capacitors for 22nm high volume manufacturing," in *2012 IEEE International Interconnect Technology Conference*, June 2012, pp. 1–3.

[23] C. H. Lin et al., "High Performance 14nm SOI FinFET CMOS Technology with 0.0174 μm^2 Embedded DRAM and 15 Levels of Cu Metallization," in *2014 IEEE International Electron Devices Meeting*, Dec 2014, pp. 3.8.1–3.8.3.

[24] J.-H. Lee, "Bulk finfets: Design at 14 nm node and key characteristics," in *Nano Devices and Circuit Techniques for Low-Energy Applications and Energy Harvesting*, C.-M. Kyung, Ed. Dordrecht: Springer Netherlands, 2016, pp. 33–64.

[25] E. J. Fluhr et al., "5.1 POWER8™: A 12-core Server-Class Processor in 22nm SOI with 7.6Tb/s Off-Chip Bandwidth," in *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, Feb 2014, pp. 96–97.

[26] G. Fredeman et al., "A 14 nm 1.1 mb embedded dram macro with 1 ns access," *IEEE Journal of Solid-State Circuits*, vol. 51, no. 1, pp. 230–239, Jan 2016.

[27] J. W. Cooley and J. W. Tukey, "An algorithm for the machine calculation of complex Fourier series," *Mathematics of Computation*, vol. 19, pp. 297–301, 1965.

[28] P. Brauer, M. Lundqvist, and A. Mällo, "Improving latency in a signal processing system on the epiphany architecture," in *Proceedings of PDP 2016*, Feb 2016, pp. 796–800.

[29] S. Hall, "FFT Optimizations for GPU and Many-Core Architectures," AUT University, Tech. Rep., 2016.