SANDIA REPORT
SAND2015-1318
Unlimited release
Printed February 2015

Scaling to Nanotechnology Limits with the PIMS Computer Architecture and a new Scaling Rule

Erik P. DeBenedictis

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

SAND2015-1318
Printed February, 2015

Scaling to Nanotechnology Limits with the PIMS Computer Architecture and a new Scaling Rule

Erik P. DeBenedictis
Sandia National Laboratories
P.O. Box 5800
Albuquerque, New Mexico 87185-MS 1319

Abstract

We describe a new approach to computing that moves towards the limits of nanotechnology using a newly formulated scaling rule. This is in contrast to the current computer industry scaling away from von Neumann's original computer at the rate of Moore's Law.

We extend Moore's Law to 3D, which leads generally to architectures that integrate logic and memory. To keep power dissipation constant through a 2D surface of the 3D structure requires using adiabatic principles.

We call our newly proposed architecture Processor In Memory and Storage (PIMS). We propose a new computational model that integrates processing and memory into "tiles" that comprise logic, memory/storage, and communications functions. Since the programming model will be relatively stable as a system scales, programs represented by tiles could be executed in a PIMS system built with today's technology or could become the "schematic diagram" for implementation in an ultimate 3D nanotechnology of the future.

We build a systems software approach that offers advantages over and above the technological and architectural advantages. First, the algorithms may be more efficient in the conventional sense of having fewer steps. Second, the algorithms may run with higher power efficiency per operation by being a better match for the adiabatic scaling rule. The performance analysis based on demonstrated ideas in physical science suggests 80,000× improvement in cost per operation for the (arguably) general purpose function of emulating neurons in Deep Learning.

Table of contents

Abstract 7
1. Introduction 9
2. Overview of a solution 11
3. A scaling rule for the third dimension 12
Energy efficiency of logic families 12
Optimal clock rate 14
Optimal Adiabatic Scaling 15
3D manufacture and distant future vision 17
Progression over time 18
4. Scaling scenarios 21
Scenario 1: Moore’s Law and personal computers 21
Scenario 2: Brain development 23
5. PIMS architecture 25
Universal nanotechnology language 25
PIMS as an assembly of tiles 26
Complexity theory and PIMS 28
Programmability 30
6. Implementing a PIMS engine 32
PIMS module implementation 34
7. PIMS computer system organization 36
Integrating storage 37
Limited parallelism for both storage and processing 37
Optimization of storage format 38
Operating system for PIMS 39
8. Derivation of PIMS tile behavior 40
Dense vector-matrix multiply starting point 40
Sparse vector-matrix multiply 43
Transformation to PIMS program 44
Input and output 45
PIMS execution 46
PIMS compiler and automatic placement 48
Instruction set architecture development 50
ALU logic design 53
9. Control system 55
Interbank architecture 56
Function layout and execution 56
PIMS subroutine execution process 57
General execution 57
10. Sparse matrices for neurons and meshes 60
Power analysis 60
Memory Access Energy 61
Compute Energy 62
System Energy 63
11. Conclusions 66
12. References 68

List of figures

Figure 1: Energy/op vs. speed tradeoff for various logic families.................................. 12
Figure 2: Scaling scenario.. 15
Figure 3: Vision of 3D scaling.. 18
Figure 4: Adiabatic system performance.. 19
Figure 5: Kryder’s Law: storage growth rate vs. Moore’s Law 22
Figure 6: Relationship between synapses and neurons.. 24
Figure 7: Incremental progress versus approach to goal 25
Figure 8: PIMS hardware-software tie.. 27
Figure 9: Complexity of a PIMS algorithm 29
Figure 10: Current supercomputer scaling (from [Mountain 14])................................... 31
Figure 11: Adiabatic clocking for four PIMS replication units....................................... 32
Figure 12: PIMS emulation... 33
Figure 13: Processor-In-Memory-and-Storage (PIMS) implementation......................... 34
Figure 14: Scaling scenarios 36
Figure 15: Systolic array.. 40
Figure 16: Diagram of the spreadsheet for 4×4 vector-matrix multiplication................. 42
Figure 17: Sparse matrix.. 44
Figure 18: Reorganization to a PIMS layout 45
Figure 19: PIMS operation chart for vector-matrix multiply... 47
Figure 20: GraphViz applied to the sparse matrix example 49
Figure 21: Instruction format.. 50
Figure 22: ISA-based PIMS emulator... 51
Figure 23: Worksheet for developing instructions... 52
Figure 24: PIMS instruction set for example... 53
Figure 25: PIMS test program... 53
Figure 26: Logic diagram of replication unit... 54
Figure 27: Control system and subroutines 55
Figure 28: Loop strategy for PIMS.. 58
Figure 29: One of 21,800 chips in a brain-sized structure... 61

List of tables

Table 1: Scaling rules from [Theis 10], with OAS on right.. 16
Table 2: Synapses and neurons for a representative biological sequence 23
Table 3: Power/energy for various combinations of technologies.................................. 64

Scaling to Nanotechnology Limits with the PIMS Computer Architecture and a new Scaling Rule

Erik P. DeBenedictis, February 25, 2015

Abstract

We describe a new approach to computing that moves towards the limits of nanotechnology using a newly formulated scaling rule. This is in contrast to the current computer industry scaling away from von Neumann's original computer at the rate of Moore's Law.

We extend Moore's Law to 3D, which leads generally to architectures that integrate logic and memory. To keep power dissipation constant through a 2D surface of the 3D structure requires using adiabatic principles.

We call our newly proposed architecture Processor In Memory and Storage (PIMS). We propose a new computational model that integrates processing and memory into "tiles" that comprise logic, memory/storage, and communications functions. Since the programming model will be relatively stable as a system scales, programs represented by tiles could be executed in a PIMS system built with today's technology or could become the "schematic diagram" for implementation in an ultimate 3D nanotechnology of the future.

We build a systems software approach that offers advantages over and above the technological and architectural advantages. First, the algorithms may be more efficient in the conventional sense of having fewer steps. Second, the algorithms may run with higher power efficiency per operation by being a better match for the adiabatic scaling rule. The performance analysis based on demonstrated ideas in physical science suggests 80,000× improvement in cost per operation for the (arguably) general purpose function of emulating neurons in Deep Learning.

1. Introduction

The concept of computing experienced an inflection point during WW II [Nordhaus 07]. Prior to WW II, computers comprised humans operating tools such as pencil and paper, abacus, etc. This meant compute power was limited by the speed and capacity of the human operator. Humans started to make programmable computers that could operate beyond the limits of a human operator during WW II. The von Neumann architecture was preeminent in these designs [von Neumann 45] as it allowed complex applications to be represented as software stored very compactly in memory, allowing applications to scale up in size. The economic activity from the applications drove technology development of the underlying electronics and enabled exponentially more capable computers over time in a trend that continues to the present day.

In 1965 Gordon Moore [Moore 65] subjectively described what later became Dennard Scaling [Dennard 74]. Moore projected exponential growth of various integrated circuit properties from 1965-1975 (but the growth continued well beyond 1975). The microprocessor was invented in 1970 (so it cannot be seen as inspiration for Moore's Law) and has over time supported the exponential path of the semiconductor industry. Moore's Law has now merged with the overall growth of computing [ITRS YY] and is nearly synonymous today (but this document will exploit the distinction).

There is a growing realization that the Dennard scaling that defined the most general form of Moore's Law has already ended. However, it is possible that the computer industry will find an alternative to CMOS and the von Neumann architecture that will continue the larger technology trend that started in WW II.

Energy efficiency is currently seen as the limiting attribute of computer technology. The first two of the attributes Moore mentioned in his article [Moore 65] were device dimension and speed. Even though growth in the first two dimensions will end soon, the number of devices should continue to grow into the third dimension at reasonable cost [Fujisaki 13] for quite some time. Speed is being addressed with parallelism, although it results in rising throughput that is somewhat less versatile than rising speed. Moore's third attribute was power per unit area, which has ballooned. Eventually, this will cause new applications that are inevitably more complex to consume too much energy to be affordable to the user. This will stymie the business model that funds the industry and create a crisis.

Industry and government are responding in different ways.

While industry partially retains the goal of straight information processing throughput [ITRS YY] for servers, it is expanding to also include mobile devices such as smartphones and the Internet of Things. These new businesses may be profitable but will not raise computer capability.

Government focuses on other goals. Supercomputers up to about an Exaflops are under development and there is government interest in similarly sized data centers. These would continue the computer performance growth trend. However, other government agencies are pressing for neural networks or artificial intelligence. These applications are quite different, but also require high throughput components.

Adiabatic and reversible computing represent technology approaches that offer the possibility of computing at "arbitrarily low energy levels." The approaches involve passing signals from one logic stage to another with loss of only a small fraction of the signal energy each time. This is in contrast to current (CMOS) logic, where all signal energy is turned into heat and regenerated from the power supply between each logic stage. Adiabatic computing has been experimentally demonstrated as cutting energy consumption around 10-100× [Karakiewicz 12], although the number of devices per gate is greater than CMOS. For the purposes of this discussion, reversible computing builds upon adiabatic computing. After significant gains in power efficiency have been realized through adiabatic computing, reversible computing could offer even more gains. In spite of theoretical promise, neither adiabatic nor reversible computing has received much attention (and this document will explain the reason).

Memory and data storage have evolved in accordance with Moore's Law as well, yet activity in these areas continues at full speed. On the forefront is a shift from moving media (disks and tapes) to technologies that use chips physically similar to those in processors but with novel devices such as Flash, Resistive RAM (ReRAM, e. g. memristors), Phase Change Memory (PCM), and Spin Torque Memory (STM). These technologies are being packaged in 3D, which allows continued growth in device complexity. The quality metric of a memory is dominated by storage capacity rather than energy efficiency, so the issues with power consumption problems that apply to processors apply only weakly to memories.

There is an unanswered question about whether a new Moore's Law-like phenomenon could develop. The idea that a computer could be made from devices at the atomic scale was popularized by Richard Feynman [Feynman 60]. Various aspects of the projected atomic scale systems can be realized individually. Biology exploits placement of molecules (albeit perhaps not atoms) to create neurons, which are unquestionably successful computational devices. We can create one or a few computing devices by placement of individual atoms [Simmons 05]. We can also create DNA that has the effect of high-volume molecular-level manufacturing when the engineered DNA replicates in living cells. Ideas for molecular or atomic-level manufacturing of computers have been proposed [Drexler 92], but are not currently practical. While we cannot do all of the activities above at once, it might be wise to consider it as a contingency because the path appears theoretically possible and many researchers are looking for it.

2. Overview of a solution

This document will present an approach to computing that involves four interrelated ideas. The connections between the ideas will be described here very briefly as a way to guide the reader.

The first idea is a scaling rule we call Optimal Adiabatic Scaling (OAS) that increases the power efficiency of semiconductors without relying on the traditional Moore's Law. The scaling rule assumes that devices are completely unchanging over time, but can be manufactured at lower cost over time. In OAS, device count goes up quickly due to lower cost while the clock speed is adjusted to go down slowly. If throughput is the product of device count and clock rate, throughput goes up. However, OAS has been formulated to cause power efficiency to rise so system power remains constant.

The second idea is to define a third-dimensional Moore's Law. Moore's Law was about scaling in the two dimensions parallel to the surface of an integrated circuit. Call the dimensions X and Y. Scaling in X and Y is approaching physical limits, but the Z dimension is starting to be tapped. The third-dimensional scaling rule is for additional features in the Z dimension, typically layers. Power efficiency must increase as the number of features in the Z dimension increases to avoid overheating the chips, which is the tie to the first idea.

The third idea is a new architecture that works with the first two ideas. Applying the first idea to the von Neumann architecture would result in computers with progressively slower clock rate over time, which would reduce performance. However, the performance of integrated Processor-In-Memory (PIM) architectures typically grows with the product of device count and clock rate. We define a minor variant on PIM, which we call Processor-In-Memory-and-Storage (PIMS).

The fourth idea is to create a computer system around the first three ideas. The document proposes a general computing system, but programs an artificial neural network as an example. A conceptual performance estimate is offered, showing improvements in power efficiency in the range of 80,000× over a current consumer-grade Graphic Processing Unit (GPU).

3. A scaling rule for the third dimension

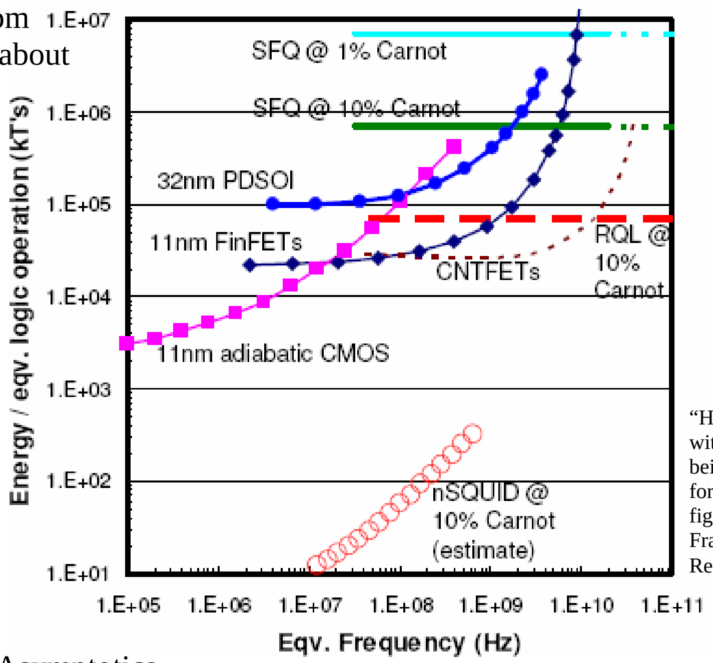
The computer industry started to actively manage the tradeoff between speed (clock rate) and energy efficiency about a decade ago, a generalization of which can lead to a scaling rule for semiconductors scaling into the third dimension.

Energy efficiency of logic families

The energy efficiency of most logic families varies by operating speed, as illustrated in Figure 1 [Frank 2014]. The curves illustrated in Figure 1A were developed with by simulation and measurement, yet the curves follow an underlying asymptotic pattern shown in Figure 1B.

A. Simulation

Note: kT at room temperature is about 4 zeptojoules or 4×10^{-21} j



"Hi Erik,/Yes, I'm ok with the viewgraphs being public, so it's ok for you to use the figure./David J. Frank/IBM Watson Research Center…"

B. Asymptotics

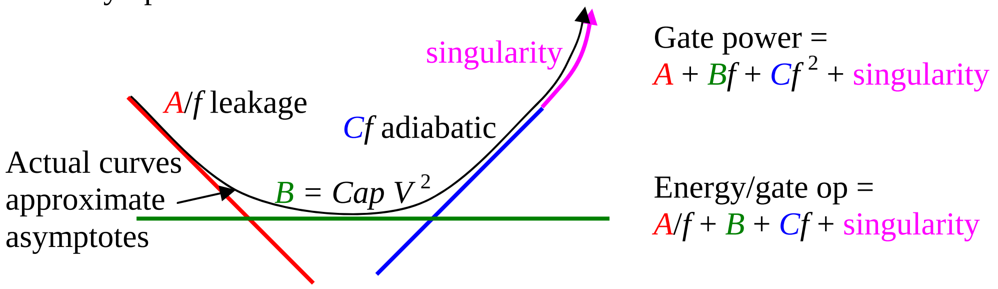

singularity
A/f leakage
Cf adiabatic
Actual curves approximate asymptotes
B = Cap V²
Gate power = A + Bf + Cf² + singularity
Energy/gate op = A/f + B + Cf + singularity

Figure 1: Energy/op vs. speed tradeoff for various logic families.

Many logic families have static power consumption, often resulting from leakage current. We will represent this power as A watts per gate (we will henceforth apply color to parameters A, B, and C consistently to distinguish them from other uses of the same letter). For a gate performing f operations per second, this term will correspond to A/f

joules per gate operation. On the log-log scale of Figure 1, this will be an asymptote of slope -1 (which would appear in Figure 1A if the graph extended further to the left).

The analysis in this document will group static power consumption with manufacturing cost. Say a user purchases a microprocessor with an expected lifetime of 3 years for $50 where leakage current contributes 10 watts to power dissipation. Assuming the user leaves the chip running all the time, buying the chip will obligate the user to pay $10/year to the power company (10 watts of power dissipation is about $10/year at nominal electric power rates). Henceforth, we will just assume the chip costs $80 ($50 purchase cost + 3 years $\times$ $10/year power cost).

The prevalent energy component in today's logic are typically called "CV^2" losses, which we will call $Cap\ V^2$ because we are using the variable C for a different purpose. These losses come from the charging and discharging of signal lines approximately once per clock cycle. These signal lines have some aggregate capacitance to ground and are charged back and forth between ground and the supply voltage once each clock cycle. This produces $B = Cap\ V^2 f$ power. We will represent this power as Bf watts per gate and it will correspond to an energy per operation of B joules per gate operation. It would be a horizontal line in Figure 1, as shown.

Some logic families may have an adiabatic (discussed below) region of Cf joules per gate operation or a power of Cf^2 watts per gate. If this behavior is present, it is represented by the blue line of slope 1 in Figure 1. The adiabatic logic families in Figure 1A are "11 nm adiabatic CMOS" and "nSQUID…", which are readily observed to have nearly unit slope over several orders of magnitude of frequency.

There is a significant literature on adiabatic and reversible computing, to which the interested reader is directed. The physical principal of a linear rise in energy per gate operation as a function of speed appears to have first been described in [Bennett 73, p. 531] and more generally described in [Feynman 96, pp. 167-172]. The specific use of the term "adiabatic" comes from adiabatic capacitor charging and apparently appears first in [Koller 92] in the context of practical circuits. This is not the same adiabatic principle as in adiabatic quantum computing, yet seems closely related to an adiabatic process in thermodynamics related to expansion and contraction of gasses.

Real logic families have a "top speed" above which device operation begins to fail, illustrated by the fast-rising curve labeled singularity. This behavior does not have an easily explained source, but is based on the well-known practice of "increasing the supply voltage to get the device to run a bit faster."

A key idea for this work is present in Figure 1. Industry has been concerned about the energy efficiency vs. frequency tradeoff since the shift from fast single-core microprocessors to dual-core microprocessors running at lower clock rate around 2003. However, this tradeoff is being performed with CMOS circuits, which cause the tradeoff to occur in the uncontrolled singularity region. This document will show how to make this tradeoff over a larger range by using adiabatic logic families.

Optimal clock rate

In this section, we formulate a new scaling rule based on a shifting energy efficiency vs. frequency tradeoff as manufacturing costs decline. To do this, we first find the economically optimal clock frequency and then apply it to the derivation of the OAS scaling rule. Note that this analysis applies only where the energy versus frequency curve in Figure 1 is close to unit slope. For non-adiabatic circuits, this region is short and our analysis primarily serves as an illustration of why semiconductor scaling is in a crisis. However, these unit slope regions can be quite long for adiabatic circuits, motivating the use of adiabatic technology as a path forward.

The cost to purchase a computer is about the same as the energy cost to power the computer over its lifetime – or at least these costs are close enough that they need to both be considered in an optimization exercise. The minimum cost per computer operation can be represented as

$\min_f (\$_{purchase} + \$_{energy})/Ops_{lifetime}$,

where f is the clock rate and the other terms are as defined as follows. Let us say the chips cost $\$_{purchase} = A$ to purchase, using the same color scheme as Figure 1B (but per chip rather than per gate). Where the curves in Figure 1 have slope 1, $\$_{energy} = Cf^2$ is the energy cost to operate the chip over the lifetime of the computer. This will capture the frequency dependence of the energy cost. The total number of ops over the machine's lifetime will be proportional to f, so we can say $Ops_{lifetime} = Df$, where D is a constant factor.

We have now transformed the problem into finding f that minimizes $(A + Cf^2)/(Df)$. To find the optimal value of f, we would set the derivative with respect to f to zero.

$d/df\ (A + Cf^2)/(Df) = 0$.[1]

By simple calculus, this has the solution $f = \sqrt{(A/C)}$.

A simple economic principle emerges when we substitute the optimal value of f into the expressions:

$\$_{purchase} = A$, as before and

$\$_{energy} = Cf^2 = C\,(\sqrt{(A/C)})^2 = A$.

Thus, $\$_{purchase} = \$_{energy}$ and the economic principle is that economically conscious users should pick their clock rate such that they spend the same amount of money buying the chips as they spend powering them!

[1] Lossy adiabatic circuits could be considered here, but this is deemed beyond the scope of this document. Say that the device energy includes a Bf term, leading us to find f that minimizes $(A + Bf + Cf^2)/(Df)$. It is readily apparent that differentiation will cause the B term to go away. The optimal frequency is thus independent of losses due to this term, but energy efficiency will decline at the optimal frequency.

Optimal Adiabatic Scaling

The economic principle in the previous section leads to the semiconductor scaling rule we call OAS. Say that we have a computer that is operating at the optimal clock rate according to the optimization in the previous section, as illustrated in Figure 2. The chip maker offers us upgraded chips with 4× as many of the same devices for the same cost per chip – in other words, a 4× reduction in cost per device. What do we do? If we simply put the new chip in the socket intended for the previous generation, the system will consume 4× as much power and move away from the optimal balance of purchase price and energy cost.

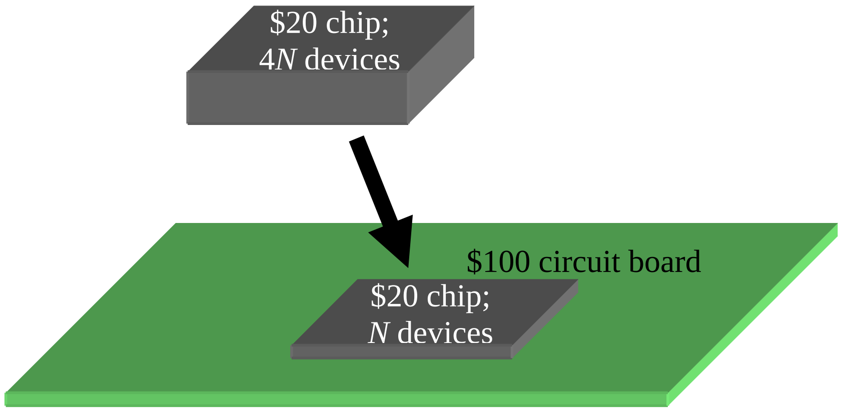


Figure 2: Scaling scenario

The solution is to turn the clock down by a factor of $\sqrt{4}\times = 2\times$. With the clock turned down, power consumption will be reduced by $2^2\times = 4\times$ from the higher level. In fact, the per-chip power consumption will be exactly the same as the previous generation. This means the new chip will fit in the old board without changing the power supplies and heat sinks. The system will also be returned to the optimal clock rate that minimizes the cost per operation, so this process can be repeated.

By turning down the clock rate, system throughput will be lower than it might have been, but there would be 4× as many devices running at ½× the speed. This will still yield a 2× increase in throughput.

The scaling above is like Moore’s Law, but not as strong. The original Moore’s Law scaled in the two dimensions parallel to a chip’s surface, but there is only one dimension perpendicular to a chip’s surface for the layer count to grow. All rates being equal, we should expect the growth rate in the third dimension to be half that of the original two dimensions.

A scaling rule results if the specific factor of 4 used in the discussion is replaced by a continuous variable N and squares and square roots are applied consistently. The factor α in typical semiconductor scaling refers to line width so the number of devices grows with α^2. To reduce confusion, we will say $N = \alpha^2$ and clarify that N represents device count.

Table 1 comprises the table of scaling rules from [Theis 10] with OAS on the right. Ignore the references to layers (which will be discussed later).

Table 1: Scaling rules from [Theis 10], with OAS on right

	Const field	Constant V				OAS (new)
		Max f	Const f	Const f, N_{tran}	Multi core	
L_{gate}	$1/\alpha$	$1/\alpha$	$1/\alpha$	$1/\alpha$	$1/\alpha$	$L = 1$*
W, L_{wire}	$1/\alpha$	$1/\alpha$	$1/\alpha$	1	$1/\alpha$	(layers) $N = \alpha^2$†
V	$1/\alpha$	1	1	1	1	1
C	$1/\alpha$	$1/\alpha$	$1/\alpha$	1	$1/\alpha$	1
$U_{stor} = ½CV^2$	$1/\alpha^3$	$1/\alpha$	$1/\alpha$	1	$1/\alpha$	$P_g = 1/\sqrt{N} = 1/\alpha$‡
f	α	α	1	1	1	$1/\sqrt{N} = 1/\alpha$
N_{tran}/core	α^2	α^2	α^2	1	1	1
N_{core}/A	1	1	1	1	α	$N = \alpha^2$
P_{ckt}	$1/\alpha^2$	1	$1/\alpha$	1	$1/\alpha$	$1/\sqrt{N} = 1/\alpha$
P/A	1	α^2	α	1	1	1§ (clarified)
$f\,N_{tran}\,N_{core}$	α^3	α^3	α^2	1	α	$\sqrt{N} = \alpha$

* Term redefined to be line width scaling; 1 means no line width scaling
† Term redefined to be the increase in number of layers; previously was 1 for no scaling
‡ Term redefined to be heat produced per step. Adiabatic technologies do not reduce signal energy, but "recycle" signal energy so the amount turned into heat scales down
§ Term clarified to be power per unit area including all devices stacked in 3D

It is necessary to redefine some terms that are specifically relevant to the first two dimensions. We make the following natural adaptations to 3D: $N = \alpha^2$ is defined to be the scaling of the number of features in the third dimension assuming no scaling in the first two dimensions. N is in lieu of L_{gate}, W, and L_{wire}, which no longer change. We clarify that power is measured from the 2D face of a 3D structure, including power from all the devices that may be stacked on top of the 2D surface. The other adaptation is that U_{stor} is defined as the energy in a signal under the presumption that signal energy is fully dissipated on every gate operation. This presumption ties the devices to the choice of circuit. So we redefine the term to be P_g, the heat created by each gate operation.

Let us express OAS in words for clarity: Say you have a chip comprised of devices that could be represented as a curve in Figure 1. To the extent that the device's curve is close to slope 1 and the chip can be manufactured with more N times as many devices but at the same cost, OAS specifies that the clock rate should be reduced to $1/\sqrt{N}$. The resulting chip will have the same purchase cost, power dissipation, and will fit in the circuit board for the old system, but will have $\sqrt{N}$ more throughput.

OAS scales further than the other entries in Table 1. There is a realization that scaling is likely to end for both voltage and capacitance (3rd and 4th parameters in the left column of Table 1). Of the columns in Table 1, only "Const f, N_{tran}" and the new OAS will apply once V and C stop scaling. The final row in Table 1 labeled "$f\,N_{tran}\,N_{core}$" represents throughput. To the extent [Theis 10] is authoritative, industry predicts no growth in throughput once *V* and *C* stop scaling. OAS continues to predict an increase.

3D manufacture and distant future vision

OAS fits nicely with current trends in semiconductor manufacturing, although progress seems destined to slow. There is growing realization that line width scaling will come to an end in a few more generations. Along with this realization, a lot of attention is being given to 3D chips.

Progress on 3D chips that are close to production and sale include both stacking of perhaps a dozen complete chips [HMC 14] and construction of chips that include the layering of up to 100 features in the vertical direction [Fujisaki 13]. By multiplying a dozen by 100, these current plans could yield chip-stacks or modules with perhaps 1200 features in the third dimension. On a geometric scale, 1200 features in one dimension is maybe halfway to a brain or the vision of a 3D computer as put forth by [Drexler 92]. Let us assume progress continues.

The number of layers could grow with a new Moore's Law-type phenomenon, although it seems unlikely that device count growth will match rates under Moore's Law. Moore's Law projected a doubling of device count about every 18 months due to a $1/\sqrt{2}$ change (reduction) in line width. While we lack rigorous data on the following statement, matching this rate with 3D layering would require a doubling of the number of layers each 18 months. This seems unlikely, but maybe half the rate is feasible.

If we apply OAS to 3D scaling, we could create a type of end-game vision as illustrated in Figure 3. As this document is written, the largest DRAM chips are 8 GBits, about 10^{10} transistors, or nominally a square array of $10^5 \times 10^5$ devices. If we make the straightforward extension to 3D, we end up with 10^{15} devices of 100 nm^3. This would correspond to $\alpha = \sqrt{10^5} \approx 300$ in Table 1.

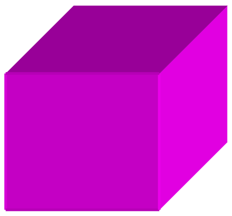
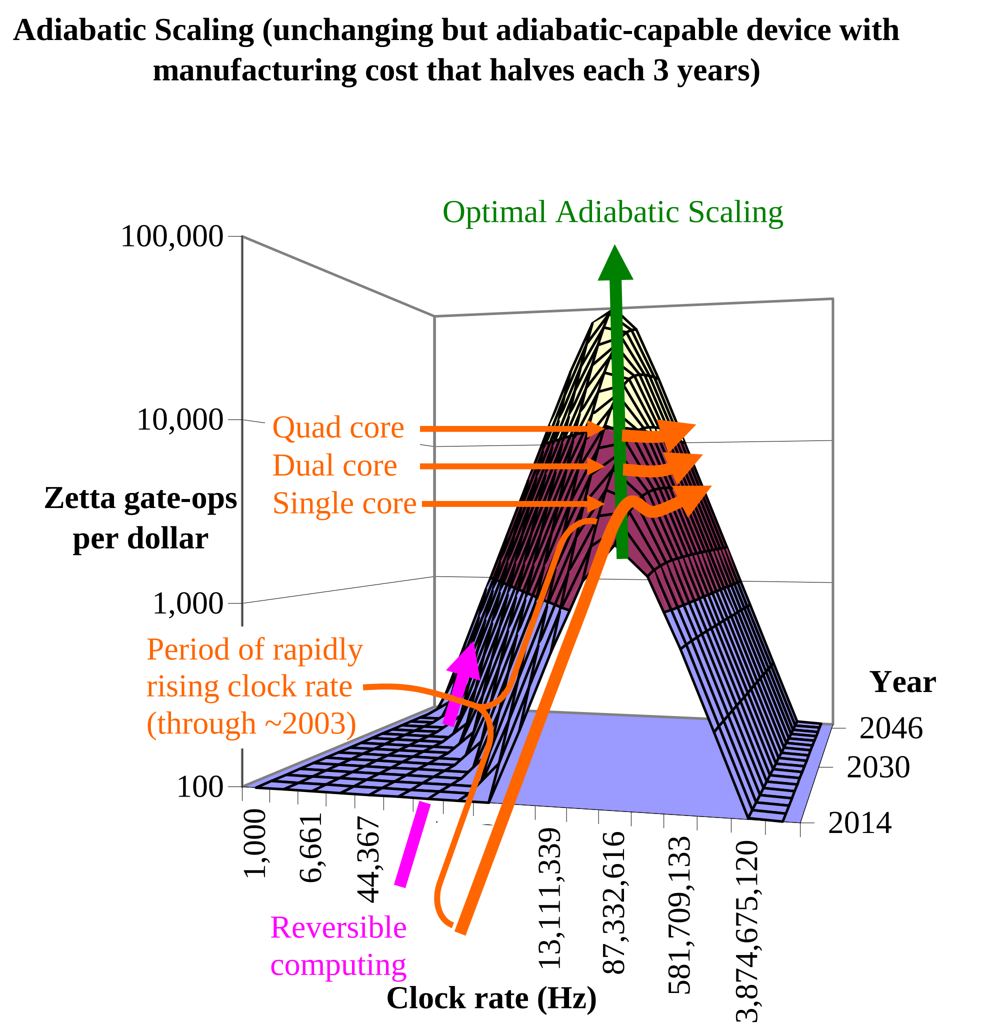
Timeframe	Today	Changes	End-Game Vision
Integration scale	10^{10} logic transistors	$\alpha = \sqrt{10^5} \approx 300$; $N = 10^5$	10^{15} logic transistors
Clock speed	3 GHz	$1/\alpha \approx 1/300\times$ (slower)	10 MHz
Performance	Chip is 2D comprised of 100 nm^2 gates.	$\alpha \approx 300\times$ reduction in joules/op OR $\alpha \approx 300\times$ increase in energy efficiency	Chip is 3D comprised of 100 nm^3 gates. 
Power per cm^2	1	1	1

Figure 3: Vision of 3D scaling

The remainder of the physical system may stay the same. In OAS, the power emerging from the bottom face (say the heat sink is connected to the bottom face) stays the same. This is consistent with the previously stated idea that the manufacturer offers a drop-in replacement with 4× as many components.

Progression over time

The ideas presented above form the basis of the plot in Figure 4. The figure shows economic value of a chip on the vertical axis, as a function of clock rate on the horizontal axis. Time progresses into the page, with the only effect of time being an exponential reduction in manufacturing cost. We warn the reader that there are too few dimensions on the printed page to illustrate both OAS and device scaling, so the diagram will not be accurate beyond the ideas explained in this document. Figure 4 is a partially calibrated plot of the economic viability ($\text{Ops}_{\text{lifetime}}$/\$) as a function of clock rate and time.



Plot based on equations developed in this document. The device must be adiabatic-capable, but with unchanging performance. However, manufacturing cost halves each 3 years.

Figure 4: Adiabatic system performance.

The arrow overlays give historical context, but the overlays are notional and not calibrated to the axes. The purchase price of computers dominated energy costs prior to about the year 2000, so the best strategy was a rapid rise in clock rate. This occurred most conspicuously during the 1990s, as illustrated by the long rising segment of the orange arrow in the front-center of the plot. The rapid rise in clock rate went over the optimal speed around the year 2000. This led to dual-core processors being introduced in about 2003, which initially had a lower clock rate (2-2.5 GHz as opposed to 4 GHz in the single-core processors they replaced) and were closer to optimal. The number of cores has trended upwards at nearly constant clock rate since. The graph in Figure 4 shows a declining clock rate (which is not correct) because the semiconductor industry continued to improve devices in this era (which is not captured in the graph).

Figure 4 shows the path to the future as well. OAS says we should follow the "ridge" into the future, as indicated by the green arrow in Figure 4. This implies a $1/\sqrt{N}$ reduction in clock rate should give a $\sqrt{N}$ increase in throughput, just as projected in Table 1.

The projected trend can be viewed as a measured blend of the original Moore's Law (orange) and adiabatic or reversible computing (magenta).

Moore's Law (in orange) has been pushing up clock rate too fast to be economically optimal. Since about 2003, Moore's Law has pushed processors over the crest of the crest of the hill (clock rate too high) and industry has responded by the architecture change of adding more cores.

Reversible computing (in magenta) currently takes power efficiency beyond the point of diminishing returns to a point where it is economically harmful. Reversibility is required to achieve an energy per gate operation below "on the order of kT" [Landauer 61]. This condition would occur (if the proper designs are used) at some specific and low operating frequency, a frequency unrelated to the manufacturing cost of the chips. The magenta arrow on Figure 4 therefore follows the plotted surface at constant frequency (a frequency picked by the author for artistic convenience). The reversible computer would have a very low clock rate and thus require a very large increase in the overall size of the computer to achieve a given throughput. Reversible computing thus appears to the far left of the graph in a region of rising but currently poor economic viability. The arrow for reversible computing will eventually intersect the arrow for optimal adiabatic computing.

4. Scaling scenarios

Throughput and device count scale in a different proportion under OAS than microprocessors scaling with Moore's Law, so we must ask what practical problems scale with OAS. We will actually find that OAS is closer to everyday problems than the scaling of microprocessors.

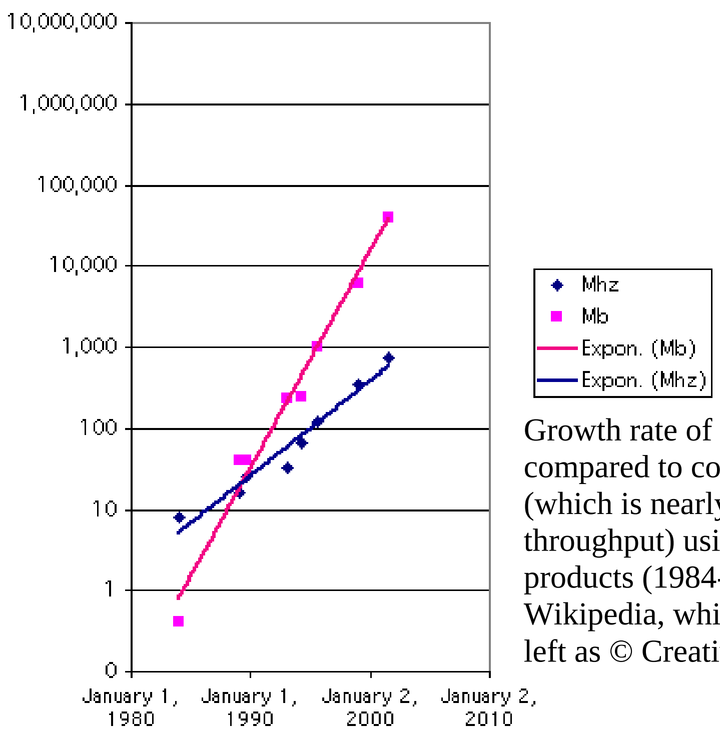
The author asserts that readers should consider computers as the intermix of logic, memory, and storage until there is a reason the think otherwise. Due to the von Neumann architecture and the semiconductor industry (which classifies chips as logic or DRAM), there may be a natural presupposition to believe future computers will be divided likewise. However, so effectively consider alternatives to the von Neumann architecture, it makes sense to leave behind the narrowing of thought processes due to old architecture.

OAS is thus considered to intimately intermix processing and memory/storage, but what is the most useful relationship between the amount of throughput versus memory/storage? This is not a familiar comparison. There is an extensive literature on algorithmic complexity theory that tells us how many arithmetic operations are required to, for example, invert an $N \times N$ matrix. Complexity theory tells us that the amount of memory scales with N^2 and the number of operations scales with N^3. Since each bit of memory requires a device, the number of devices will scale with N^2. However, complexity theory does not tell us how long we are willing to wait for the computer to complete the N^3 operations. We have to know this amount of wall-clock time to know how much throughput is needed, and thus to know whether it is reasonable for throughput to scale with the square root of the number of devices.

We will give two examples in the discussion below, each offering insight but with limitations.

Scenario 1: Moore's Law and personal computers

The first example is the historical scaling of processing speed and memory/storage in commercial products over the last few decades. In fact the storage capacity of disks has grown at about twice the rate of processor clock rate [Walter 05], as illustrated in Figure 5. This is the same as OAS, where device count growing with N (comprising devices that could be used for memory/storage) is accompanied by throughput growth of $\sqrt{N}$.



Growth rate of HDD storage space compared to computer clock rate (which is nearly identical to throughput) using Apple consumer products (1984-2001). From Wikipedia, which cites the diagram to left as © Creative Commons.

Figure 5: Kryder's Law: storage growth rate vs. Moore's Law

Growth is due to something other than algorithmic complexity (for which the author does not have a reference). Computers run applications that evolve as the underlying system grows, such as word processing or supercomputer simulations. Let us use word processing as an example. In the 1970s, word processing might have comprised a 4 Mhz 8/16 bit microprocessor with a floppy disk for storage. Floppy disk capacity was around a megabyte. The document you are reading is somewhat over a megabyte, so apparently it is possible to write a document on a computer whose storage capacity is about the size of the document. However, the author does not consider his 2 GHz laptop processor with a 500 GB disk being used to write this document as "too powerful" – even though it has a 500× faster clock and a 500,000× more storage. This is because the word processing function scaled in a way not covered by algorithmic complexity. The computer setup in the 1970s was sufficiently powerful to enter a document and get the spelling, phrasing, and pagination correct. However, today's word processing setups also include the ability to do research on the topic matter and simulations. So the laptop used to write this document also has copies of all the referenced documents – and the author's document library from which the references were selected. The expansion word processing's functionality consumes much of the growth in computer power.

The author's conclusion is that computers have scaled close to the OAS rate even though microprocessors scaled with Moore's Law, which leads to the thought that OAS is a good fit for computers (as a whole) but not necessarily a good fit for the main components of a computer separately (like microprocessor and memory treated as independent objectives).

Scenario 2: Brain development

The second example is of the activity that created control systems for animals, called brains. Irrespective of whether you view this as God's project, evolution, or something else, it created a family of what we would consider to be robot controllers. It would be instructive to see how throughput and storage capacity scale for this exemplary application sequence and how it compares to OAS.

There is a difference that complicates the analysis but not the conclusion. We chose to define OAS for scaling chips at constant power per chip, but have neither discussed nor precluded systems with multiple chips. However, the brain scale-up sequence raised both size and power consumption along the evolutionary sequence. So we should expect brains to scale like a growing number of OAS chips, but with the same ratio of throughput to device count in each OAS chip.

Brains have some easily understood properties that make this analysis straightforward. It widely understood that information is stored in the synapses whereas it is the neurons that compute. Furthermore, biological studies show the power consumption is very nearly the same for neurons of all types and sizes [Herculano-Houzel 11]. (For example, neurons in the cortex are an order of magnitude larger, on average, than those in the hippocampus. Cortex neurons have more synapses and a lower activity rate, but the same wattage per neuron.) Given these simple properties, the ratio of synapses to neurons fills the same role as the ratio of devices to throughput in OAS. The numbers synapses and neurons are given in Table 2 [Wikipedia YY] for a representative biological sequence.

	Synapses	Neurons
Roundworm	7.50E+03	3.02E+02
Fruit fly	1.00E+07	1.00E+05
Honeybee	1.00E+09	9.60E+05
Mouse	1.00E+11	7.10E+07
Rat	4.48E+11	2.00E+08
Human	1.00E+15	8.60E+10

Table 2: Synapses and neurons for a representative biological sequence

To establish the dependence, we plot the number of neurons as a function of the number of synapses in Figure 6, along with the line of $N_{neurons} = ½N_{synapses}^{¾}$. The author picked the factor ½ and the exponent ¾ by subjective curve fitting and asserts the curves match.

However, brains scale with an exponent of ¾ whereas OAS has ½. Brains got bigger and more power-hungry along the evolutionary sequence whereas OAS is scoped to a single chip with constant power consumption, but this difference can be investigated.

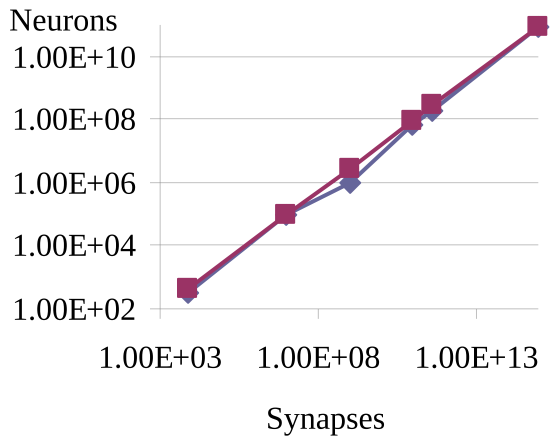


Figure 6: Relationship between synapses and neurons

The difference in exponents can be reconciled pretty closely, but a full discussion of this point is beyond the scope of this document. Say brains scaled with γ, where γ is the number of synapses (and hence its storage capacity), and $\gamma^{3/4}$ is the number of neurons. How could the brain scaling activity have used OAS chips as the underlying technology? The answer turns out to be[2] that the number of OAS chips in the brains would scale with N and the number of devices in each OAS chip would scale with N as well. This means $\gamma = N^2$. This constitutes a hybrid scaling rule. Half the scaling is through OAS and the other half through simple replication.

[2] The author does not have an extensive theory on this point, nor is such a theory essential to the conclusions of this document. The author guessed an answer and first guess worked.

5. PIMS architecture

The PIMS architecture will be described in a series of abstract steps. At a high level, the von Neumann architecture splits computing into hardware and software. At a similar level, PIMS specifies computing where the software and hardware are essentially the same, albeit different in expression at a low level. In PIMS, a simple transformation can switch between a hardware schematic diagram to software. After the concept of hardware-software duality is expressed, a more traditional architecture will be offered that essentially emulates a hardware/software specification. This approach is essential to making a claim that the architecture achieves efficient execution.

Universal nanotechnology language

Computer architectures have been the result of human preference, but PIMS is being designed to follow a natural progression towards a nanotechnology goal.

Computer architectures usually start as the architect's idea of what a computer should be like, and then get evolve based on feedback from people or workload analysis. As illustrated in red in Figure 7, Moore's Law (as applied to microprocessors) originally specified that microprocessors would become bigger and faster over time [ITRS YY]. The direction shifted around 2003 to the microprocessors having a rising number of cores at the same clock speed. Both faster clocks and more cores are suitable as measures of progress beyond von Neumann's original EDVAC computer.

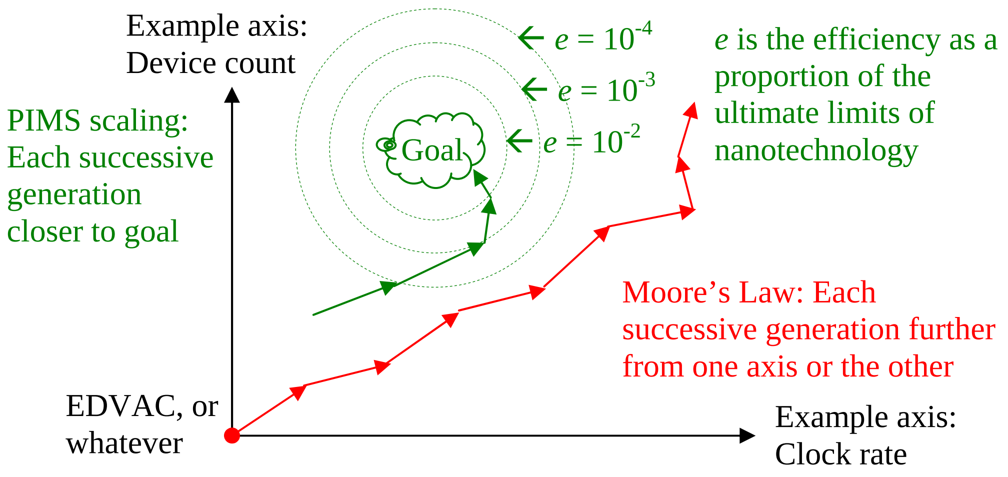

Example axis: Device count
PIMS scaling: Each successive generation closer to goal
Goal
← $e = 10^{-4}$
← $e = 10^{-3}$
← $e = 10^{-2}$
e is the efficiency as a proportion of the ultimate limits of nanotechnology
Moore's Law: Each successive generation further from one axis or the other
EDVAC, or whatever
Example axis: Clock rate

Figure 7: Incremental progress versus approach to goal

The general PIMS architecture has the goal of reaching the ultimate performance limits of nanotechnology, as illustrated in green in Figure 7. We measure closeness to the goal by a hypothetical numerical efficiency factor *e*, which is the fraction of the goal that can be achieved at a particular time. An increase of *e* from one generation to the next represents the PIMS version of progress.

There is an emerging leadership position that computing will bifurcate into two paths, with the first path carrying Moore's Law, CMOS, and the parallel von Neumann architecture to maturity while the second path devises a new device to empower a new computing concept with an exponential improvement path.

This document offers a new computing concept with an improvement path illustrated in green in Figure 7 and measured by a rising efficiency factor *e*. The reader will see in later sections that programming PIMS will include the ability to specify arbitrary nanocomputers by using the programming language as a 3D nanotechnology layout tool. The author has no idea whether the efficiency parameter *e* will rise exponentially or as some other function of time. However, any nanotechnology computing structure that can be created in the universe could be specified as a PIMS design, including the design where *e*=1. At some point, it will be possible to manufacture that design. By the construction of this argument, our improvement path will not extend past the point where *e*=1 (except through a change in assumptions, which will be discussed below as well).

PIMS as an assembly of tiles

PIMS is based on structures like 2D bitmap images or 3D assemblies of Lego blocks, but where each pixel or block represents logic, memory/storage, or interconnect between the block and its neighbors.

Let us define a practical version of PIMS for expository purposes. A PIMS structure will comprise cubic sub regions of the 3D volume of the chip in the lower-right of Figure 3, each region performing an elementary computing function. We will call each sub region a tile unless it is necessary to emphasize the 3D structure, in which case we will call it a block. The tiles will be in three classes: logic, memory/storage, and interconnect, as shown in Figure 8A. Variants of each type will be needed, such as different logic functions, memory/storage with different initial data, and interconnect that moves data in X, Y, Z directions, around corners, etc. We will call each tile containing information a memory tile unless it is necessary to emphasize the non-volatility of the data, in which case we will call it a storage tile. Figure 8A shows three different subtypes for interconnect as examples, but let us defer further discussion of subtypes until later.

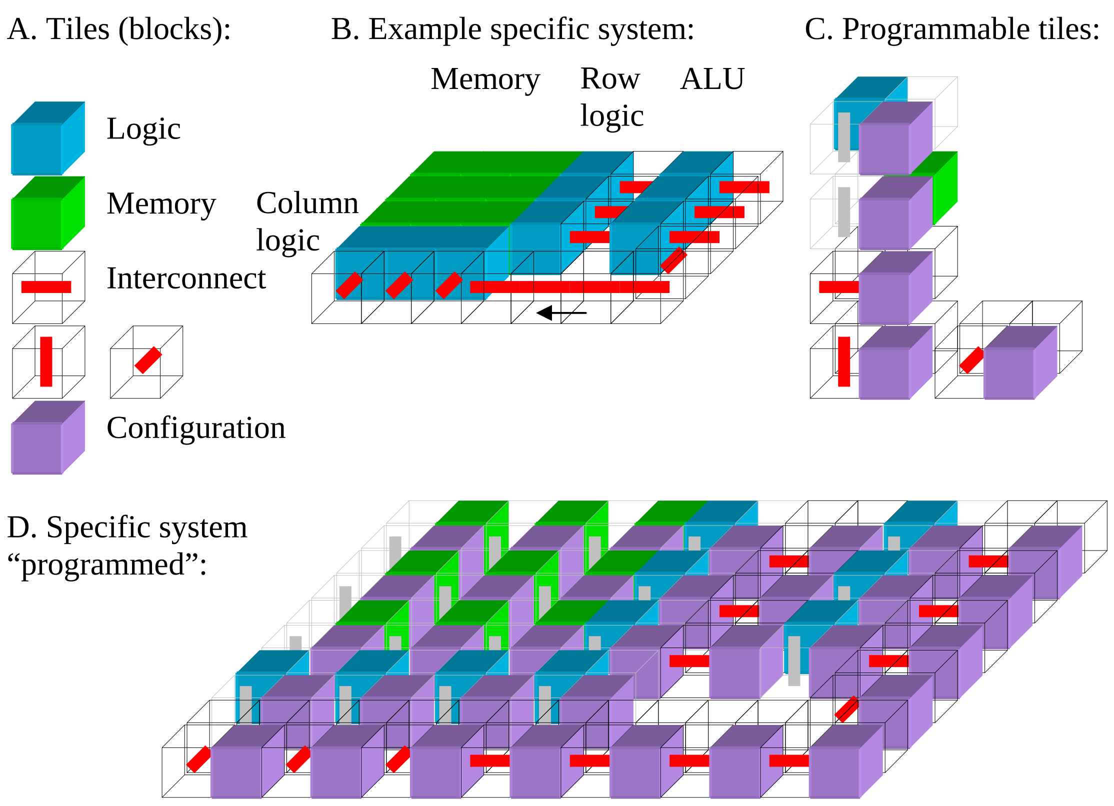


Figure 8: PIMS hardware-software tie

The architecture does not specify details of the pixels or tiles, which is deliberate decision by the author and with precedent. The von Neumann computer architecture is widely viewed as storing instructions (and data) in memory. However, two computers could both qualify as having the von Neumann architecture even if they had completely different instruction sets. While we do not propose the following for anything more than a thought experiment, PIMS tiles could represent atoms of Silicon, Copper, Oxygen, and some dopants, with a PIMS structure comprising the placement of atoms in space. Except for an issue of non-rectilinear placement, any computer can be specified in this way.

To show universality of PIMS, a von Neumann computer can be specified by a few layers of tiles that have the same organization as the physical structure of a microprocessor chip and one or more memory chips. A von Neumann computer could look like Figure 8B but with more tiles. Software loaded onto a von Neumann computer could be created by a very small adjustment to a compiler. The compiler would be changed so its output file specifies an array of initialized storage tiles, where the initial values correspond to the compiled code of a program.

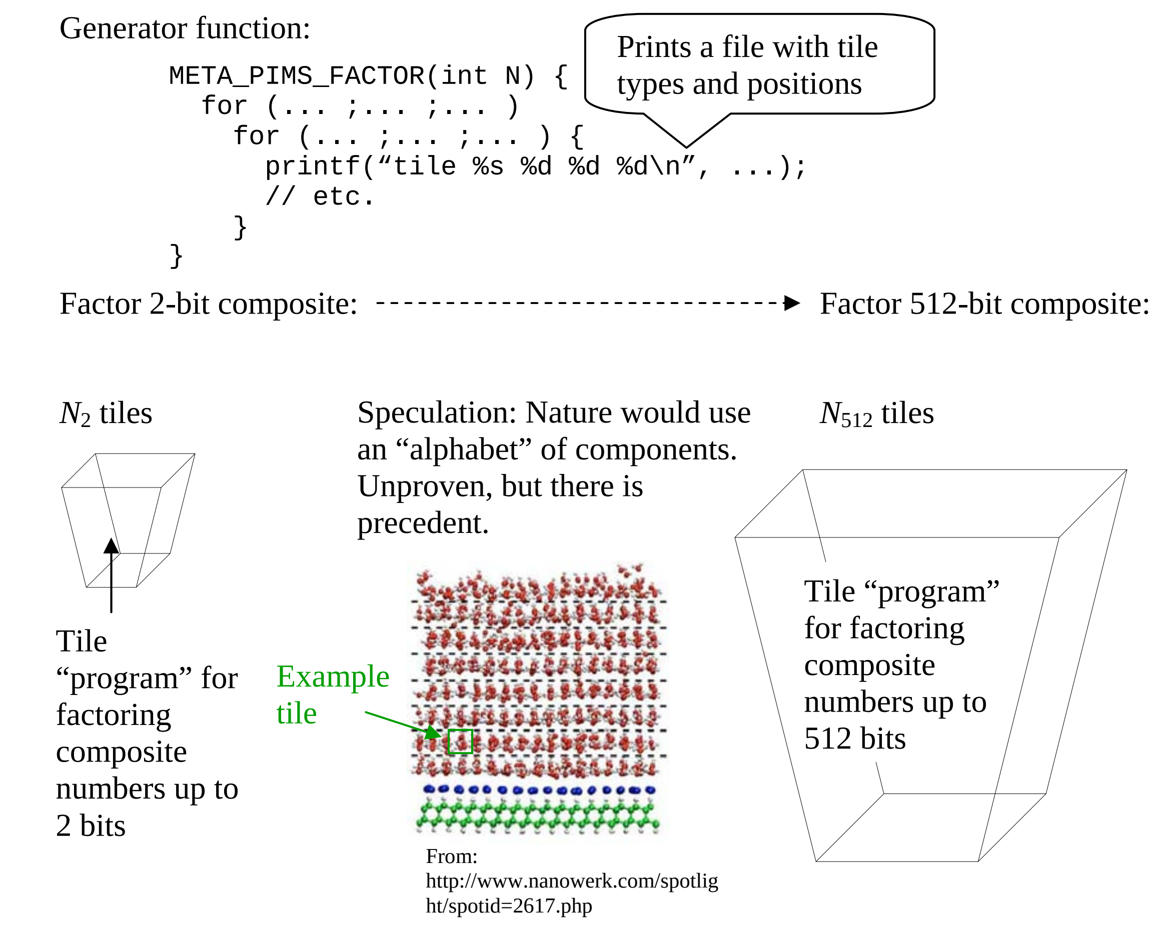
We will show arrangements of tiles later in this document that are vastly more efficient at computing than the von Neumann architecture. For example, we show how to preprocess a graph (specifically from a sparse matrix) so that the nodes of the graph become logic and memory tiles and the arcs of the graph become interconnect tiles. This example is contrary to the idea that a computer architecture has a definable form, as the computer architecture in this example would be the same as the sparse matrix pattern of the input data set.

Complexity theory and PIMS

The efficiency factor *e* discussed above is based on a new type of computational complexity theory developed here for PIMS, which offers a more physically realistic bound on algorithm runtime than is common in the field.

There is extensive theory on algorithmic complexity based on the amount of resource used to solve a problem with input size *N*. However, this theory has only a weak tie to running time and power efficiency. Consider that the operands for add and multiply need to come from a memory, where the memory may just be a single word or may be as large as the problem size *N*. Storing *K* units of data in the physical would will require a volume proportional to *K*. To access an arbitrary location would require a communications activity over distance $O(K^{1/3})$, which affects both time and energy of that access. If *M* is larger than a constant, such as $M = N$, then proper application of computational complexity theory should add a factor for the resources required to access memory. The framework of algorithmic complexity theory would support adding this factor, but it has not been done to date. Perhaps it has not been done because there is no agreement on a universal computer architecture upon which to base distance. However, the tile or block structures proposed for PIMS are a universal computer architecture that (the author claims) duplicates the physical structure of the universe sufficiently to create a useful resource metric.

The best assembly of tiles for solving a specific problem of size *N* will a distinguished structure, like a law of physics. Consider for example the problem in cryptanalysis of finding the prime factorization of an *N*-bit number. Hypothetically, there will be some optimal arrangement of atoms that will form a computer that can best solve this problem. There may be different arrangements corresponding to different optimization targets (e. g. lowest energy, shortest time to solution) and environmental domains (e. g. room temperature versus cryogenic operation, availability of all chemical elements versus specific subsets). Given a choice of the above assumptions, Figure 9 shows some example arrangements for $N = 2$ and $N = 512$. We can infer that there would be similar arrangements for all values of $N > 1$.



Generator function:

```
META_PIMS_FACTOR(int N) {
  for (... ;... ;... )
    for (... ;... ;... ) {
      printf("tile %s %d %d %d\n", ...);
      // etc.
    }
}
```

Prints a file with tile types and positions
Factor 2-bit composite:
Factor 512-bit composite:
N_2 tiles
Speculation: Nature would use an "alphabet" of components. Unproven, but there is precedent.
N_{512} tiles
Tile "program" for factoring composite numbers up to 2 bits
Example tile
Tile "program" for factoring composite numbers up to 512 bits
From: http://www.nanowerk.com/spotlight/spotid=2617.php

Figure 9: Complexity of a PIMS algorithm

We do not know how to find the optimal arrangements of atoms identified in Figure 9; it is only important that they exist. Computer scientists have identified algorithms for solving problems like sorting, matrix inversion, and so forth that minimize some set of operations, such as compares, adds, and multiplies. Figure 9 suggests defining complexity measures like lowest energy or time to solution and finding the arrangement of atoms that, when viewed as a special purpose computer, will solve a problem at minimum cost. The arrangements of atoms in Figure 9 are therefore optimal for their algorithm based on natural physical laws (given the optimization target and assumptions mentioned above) as opposed to some human preferences about the operations a computer is supposed to have.

The optimal atomic arrangement as a function of *N* could be the result of a generator function, which we could call a meta algorithm (`META_PIMS_FACTOR` in Figure 9) or maybe a "macro expander." The meta algorithm takes the parameter *N* and outputs the arrangement of blocks corresponding to the computer that would factor numbers up to *N* bits. While the runtime of the meta algorithm is not important, the resulting structure of tiles could be assessed for the number of blocks, average running time, average energy consumption, etc. when it factors a number.

Therefore, the language of the `META_PIMS` algorithm can be viewed as a language for encoding optimal algorithms in the sense of minimum run time or energy consumption. This language of the `META_PIMS` will be equivalent to the programming language for PIMS.

The author believes that the vision described in the paragraph above is overkill. It is likely that nature has structure that could be used in PIMS to simplify this vision. In other aspects of nature, such as biology, the periodic table of elements, etc., natural processes create an alphabet of components – such as DNA base pairs and elements whose chemical properties are related by the column of the periodic table. While the author offers only speculation on the following point, perhaps the optimal arrangement of atoms to factor *N*-bit numbers would be a structure built from a limited number of molecular building blocks. The center illustration in Figure 9 is author's view[3] of nanostructures that might have behaviors such as computing, memory, and interconnect. These natural structures should inspire the definition of PIMS tiles, although it is not entirely necessary that they exist for the purposes of this document.

For the purposes of what follows, the reader is asked to consider tiles illustrated in Figure 9 to be abstract just like instructions in a von Neumann computer. Diagrams of a von Neumann computer could have a big box labeled "memory" containing little boxes labeled "instruction." Defining the instruction set is left to the implementer. In the same spirit, the specific behavior of the tiles in Figure 8 is left as an exercise for the reader. However, an example later on in this document will derive the behaviors for the tiles of a sparse matrix solver.

Programmability

A computer is not limited to solving one specific problem, but there is an expectation that a computer can be programmed to solve many problems.

Usually, the architect has to choose a resource balance when constructing a programmable computer. In the discussion above, the architect would assemble logic, memory, and interconnect blocks in the quantities needed for a particular application. This ratio typically can vary dramatically from one application to another. Due to the current focus on energy efficiency, today's computers tend to be designed around the logic. [4]A common approach to energy scaling is to focus on the computing power, identify a scaling law, and demonstrate achieving a "Moore's Law" type scaling for energy efficiency. The problem with this approach is that the total system power is made up of more than just the compute power, and if those portions scale more slowly (or not at all), the power scaling expected is incorrect. [Figure 10] shows a chart where the total power is 100W at technology generation #1, and the 4 components of the power are equal in value. Assuming specific differing power scaling laws for the components, it becomes apparent that the actual power scaling is much less than the compute scaling.

[3] The graphics are inspirational only; they are just graphics of molecules from the Internet.
[4] The remainder of the paragraph and Figure 10 are courtesy of David Mountain [Mountain 14].

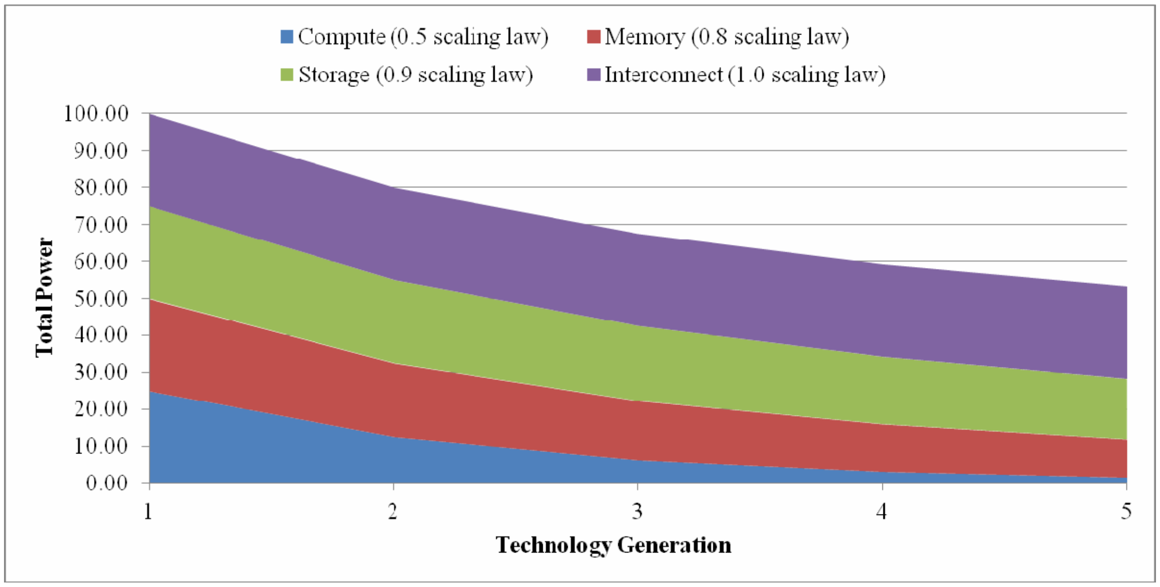


Figure 10: Current supercomputer scaling (from [Mountain 14])

PIMS addresses the balance issue by stipulating that the configurable tiles contain equal resources for logic, memory, and interconnect (where memory may be nonvolatile, corresponding to storage in Figure 10). Equality between these different resources would be through an approximate balance of manufacturing cost and power consumption. The internal structure of PIMS will therefore appear homogeneous to the user, with all tiles being equally capable of assuming logic, memory, and interconnect functions. The effect of balancing the resources will be to bound the amount of inefficiency resulting from dedicating hardware to one application versus another.

PIMS will have composite tiles or blocks as shown in Figure 8C, offering programmability at the cost of efficiency. Each composite tile or block would comprise one unit for each of the logic, memory, and interconnect functions in Figure 8A plus a new tile for configuration. The configuration tile would cause just one of the other three tiles to become active[5] based instructions from the programmer. By setting the configuration tiles correctly, these four-unit composites could emulate any specific computing device of the original type. This emulation would have efficiency losses, which is the *e* factor in Figure 7 (representing proximity to the goal). One of the losses would be *e* = ¼, as each of the original tiles is replaced by a four-tile composite. There would be other losses. Communications would have to cover longer distances, taking more time and energy.

Functionally, the use of these composite tiles would allow “emulation” of any specific structure, as shown in Figure 8D. If the reader looks carefully, each tile of Figure 8B has been replaced by a four-unit composite that performs the same function.

[5] Later on in the document, we show tile configurations that turn on all the functions at once. For the time being, let us view the act of turning on more than one function at a time as an optimization.

6. Implementing a PIMS engine

By exploiting economies of scale, we can make an efficient system for executing the four-unit configurable tiles in Figure 8C and used as shown in Figure 8D. The idea is to make a "PIMS replication unit" comprising a memory bank to hold information defining the tiles and connected to Arithmetic and Logic Units (ALUs) that perform the function specified by the tiles. A large system would comprise an array of PIMS replication units, connected so they can duplicate the behavior of a single large PIMS.

Data in the memory bank is viewed as a 3D array of words using some natural mapping of memory rows and columns to spatial dimensions. The structures in Figure 8B and Figure 9 are stored in the memory using such a mapping, one tile per word. To execute the algorithm, the tiles are accessed row-at-a-time and the ALU emulates the behavior of all the tiles in each row. The ALUs would only emulate behavior that is pertinent to computation, and not, say, parasitic thermal motion that would occur in a nanotechnology implementation.

Figure 11 shows how the behavior above can be implemented in a way that could have OAS. A typical memory bank size is about 1000×1000 bits due limitations of analog electronics and not scalable.[6] In the example illustrated, the replication unit contains one memory bank of 100,000 tiles as an array of 1000 rows by 100 tiles, each tile being a 10 bit word from 10 one-bit columns. A control unit will cycle access to the tiles by row, reading each row and (if necessary) writing it back with updated information. At this point in the exposition, it would be sufficient for the rows to be accessed in a simple row-at-a-time repeating pattern from top to bottom (but the access pattern will be extended later in the document). The rotator switch illustrated on each bank connects just one of the rows to the inductor and switches only once per cycle.

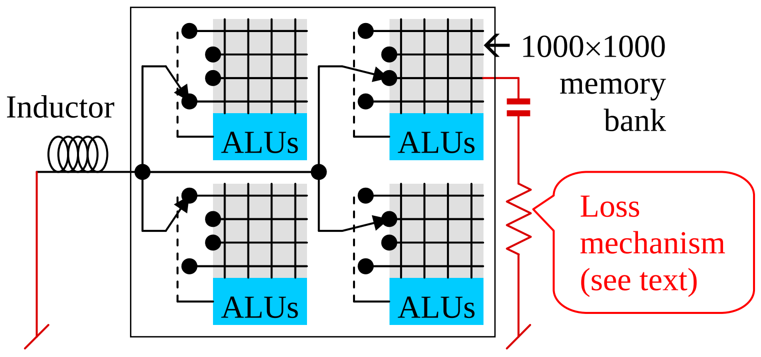


Figure 11: Adiabatic clocking for four PIMS replication units

[6] Memory banks in DRAM (for example) are around 1000×1000 bits due to the capacitive loading and delay associated with row and column wires that must span 1000 storage cells. The anticipated emergence of 3D may change this to some other size, such as $A \times B \times C$, for some other fixed values that we do not know at this time.

A regular memory access pattern is essential to energy-efficient scaling, but this is not a problem in this application. A traditional memory would turn the CV^2 energy in the row lines into heat every cycle, whereas the adiabatic circuit demonstrated in [Karakiewicz 12] recycles the CV^2 energy to increase energy efficiency. The circuit puts the collective capacitance of one memory row in each bank in series with an inductor, creating a tank circuit. It would be impractical to have more than one or a few inductors on a system, so this implies tying the rotator switches together as shown in Figure 11. Tying the switches together synchronizes the access pattern in time, preventing the random access patterns characteristic of the von Neumann architecture. This is not a problem for PIMS because the programmable behavior is in the structure of the tiles rather than in the order of accessing the tiles. As described earlier in this document, OAS is not lossless but the loss mechanism (shown as the red resistor in Figure 11) becomes more efficient as frequency is reduced.

The PIMS implementation will reorganize the view shown in Figure 8 to that shown in Figure 12, although they are functionally identical.

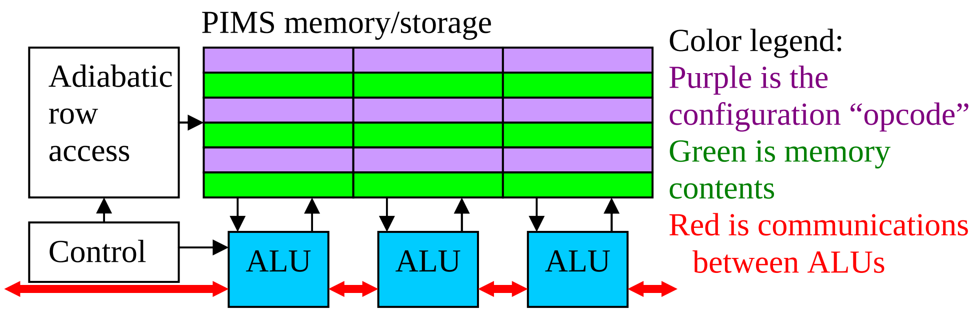


Figure 12: PIMS emulation

The purple configuration tiles in Figure 8 will become what are essentially opcodes stored in memory that define a tile's class (and subtype), as shown in purple in Figure 12.

If the cell in Figure 8 is configured to be memory, the data will be stored in the PIM memory of Figure 12 shown in green.

If the cell in Figure 8 is configured to be logic (blue), the function is actually performed using CMOS logic in the ALU.

If the cell in Figure 8 is configured to be interconnect (red), the function is also performed by the ALU. Communications to a tile adjacent in the left-right direction can be accomplished by nearest neighbor data exchange between ALUs. Communications to a tile below can be accomplished by storing data in the ALU until the next row is accessed. Communications to a tile above would require a long delay, or reversing the access pattern (which are topics discussed later in this document).

PIMS module implementation

A baseline module implementation and illustrated in Figure 13. Information defining tiles would be stored in, say 100, "PIMS 3D storage array layers A1-A100." These layers could be Flash, ReRAM (memristors), Phase Change Memory (PCM), or some other option. The layers could also be DRAM as in [HMC 14], although such a system would not have persistent storage. The ALUs could be in a CMOS chip labeled "PIMS logic," with the ALUs passing information via the red lines. A PIMS replication unit would comprise a vertical stack comprising the ALUs on the bottom with a vertical stack of memory banks layered on top. In a simple case, the specific PIMS structure could comprise one CMOS base layer (blue) and, say, 100 layers of memory/storage (purple and green). Later on, a stack of the just-described PIMS systems could be replicated, say, 10 times (9 extra copies labeled "Stacked PIMS B, C, D, E, F, G, H, I, J) with chip stacking.

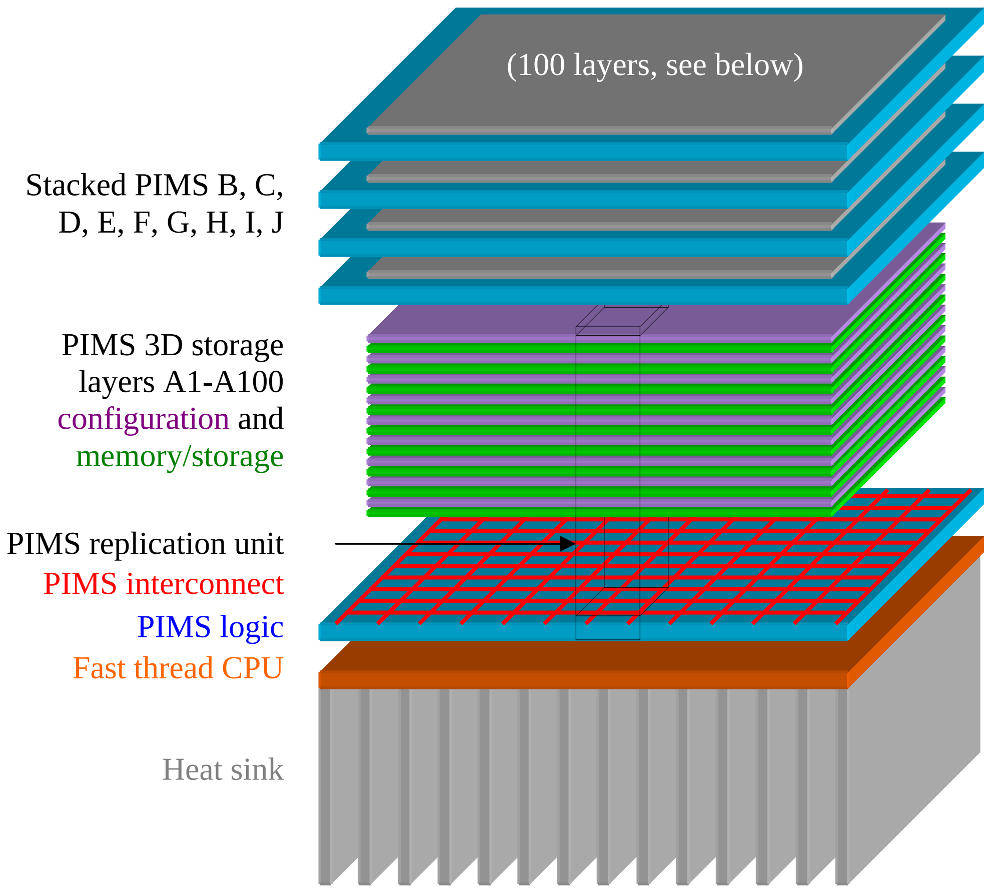

(100 layers, see below)
Stacked PIMS B, C, D, E, F, G, H, I, J
PIMS 3D storage layers A1-A100 configuration and memory/storage
PIMS replication unit
PIMS interconnect
PIMS logic
Fast thread CPU
Heat sink

Figure 13: Processor-In-Memory-and-Storage (PIMS) implementation.

Extending the ideas in the previous paragraph, the module could be a stack of (a) a heat sink capable of dissipating, say, 50W, (b) a conventional CPU (orange), (c) the PIMS system A1-A100 just described, and (d) optional additional layers B-J. The conventional CPU could be powered up alone, dissipating 50W. Alternatively, the conventional CPU would be powered down and PIMS layers A1-A100 powered up. Let us say this configuration dissipates 50W as well. In a third option, layers A-J would be powered up all at once and the clock rate reduced consistently with OAS to keep power dissipation at the same 50W.

The task of much of the rest of the document is to understand how to fill in the details of the computer just described so it can be programmed conveniently. At one level, this means specifying the details of the classes of tiles. These tiles and their subtypes will become an instruction set of sorts for arbitrary nanostructures (which we have called nanotechnology programming language). At another level, this means understanding how an entire computer of a billion or more blocks could be managed so that one portion is storing data, another is computing, and so forth.

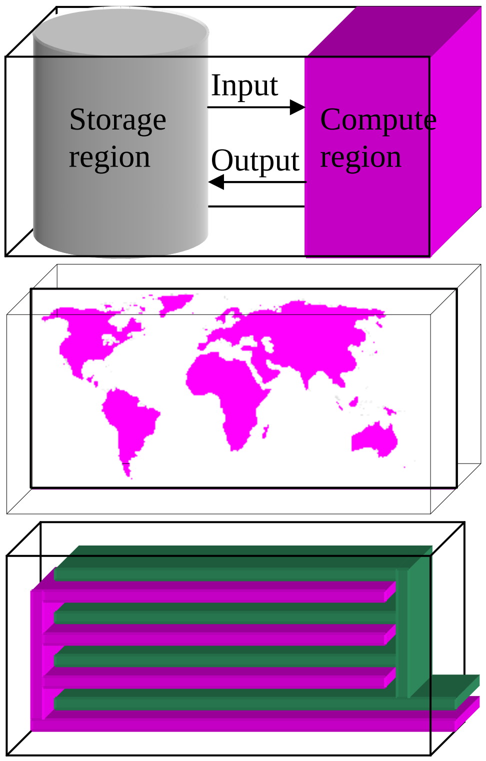
7. PIMS computer system organization

The combination of PIMS and OAS can trade off energy efficiency for speed and vice versa, potentially leading to a system that could dynamically shift between being single-threaded and fast like today's microprocessors and being highly parallel and power efficient.

The author's view is that a future user should view a PIMS system as a single 3D space containing tiles. The space may be built from multiple physical structures like Figure 13, just as a memory in a conventional computer is built from many DRAM chips. However, a user may choose to use just a subset of the space for a particular task. Properly controlling the size of the subset can alter the efficiency-speed tradeoff.

Figure 14 shows three scenarios for distribution in a PIMS system, each with advantages. Each of the large rectangular outlines represents a 3D structure of tiles as shown in Figure 3, with the structure within the outlines showing scenarios for the how storage and processing may be distributed within the structure. The scenarios will be described below, but the processing sequence and efficiency boost are summarized in the right column of Figure 14 for all the scenarios.

A. von Neumann model with input/output:



Read input
Parse
Process with √N efficiency boost
Format
Write output

B. Processor-In-Memory-and-Storage:

~~Read input~~
Parse
Process with $N^{\delta/2}$ efficiency boost
Format
~~Write output~~

C. Persistent object store of data pre processed for optimal access:

~~Read input~~
~~Parse~~
Process with √N efficiency boost
~~Format~~
~~Write output~~

Figure 14: Scaling scenarios

Integrating storage

Figure 14A is a scenario where PIMS implements the input-processing-output architecture of today's computers. The physical structure of the PIMS computer is imagined to be comprised of a sub region for storage (gray cylinder shaped like a disk drive) and a sub region for processing (magenta). Data are redistributed from their storage positions to computing positions during the "read input" phase of a program. The computation then executes with up to $\sqrt{N}$ power efficiency boost due to OAS. Data are redistributed again during output. This is the baseline scenario, where data are redistributed to suit a computation with an additive cost for the redistribution. The reader is asked to pay attention to the fact that the cost is additive, as other costs behave differently.

Limited parallelism for both storage and processing

Figure 14B is a second scenario that leads a range of behaviors – the traditional von Neumann behavior on one end of the range and the typically expected processor-in-memory behavior on the other end. This scenario arises when scaling causes data or processing to be present in a progressively smaller sub region of the overall PIMS structure.

There are some reasons for limited parallelism unrelated to PIMS. For example:

- With respect to processing, the algorithm may lack parallelism. The extreme example is that we are processing the "serial portion" of code as defined by Amdahl's Law [Amdahl 67]
- The algorithm's input may be a subset of data from a larger set, where the subset is not known in advance. For example, a program that processes the result of a database query
- The added cost of moving the data in accordance with the input/output model in Figure 14A exceeds the cost of processing it in place, even though processing in place is less efficient

While OAS will be applicable even if activity is limited to a constant fraction of the computer, like 50%, 1%, etc., further analysis will be needed if activity shrinks to a progressively smaller subset such as $\sqrt{N}$, $\log N$, or a fixed number of devices. For example, the illustration in Figure 14B is the storage of a 2D structure (a map of the Earth) within the 3D volume of a computer. Instead of storing the example map (and other data) on disk drives, imagine they are stored in the same 3D structure as will be used to process them later on. As the system scales, the storage used by the map in Figure 14B grows as $N^{2/3}$ out of the total storage N, comprising the declining fraction $N^{-2/3}$.

We will now analyze the consequence of running an algorithm on a progressively smaller subset of the PIMS architecture. Let us say the resources we can exploit scale with N^{δ}, $\delta \leq 1$. Let us then repeat the scenario of a chip maker offering an upgrade to a new generation of chip with 4× as many devices at the same cost, as shown in Figure 2. In this case, the number of active devices grows by a factor of 4^{δ}. This would cause the chip to move off the optimal cost per operation unless the clock rate was lowered. By multiplying the clock rate by $4^{-\delta/2}$, power per unit device would drop by 4^{δ}, precisely balancing the increase in number of active devices and maintaining the economic optimum. This would result in a higher speed clock but less of an energy efficiency boost than in the original explanation. Specifically, clock speed and energy consumption per operation would decline by the smaller amount $4^{\delta/2}$ rather than $4^{½}$.

OAS can degenerate into the same scaling behavior as a von Neumann computer. Say you have a completely serial algorithm that will run on one processor no matter how large the problem or computer. This corresponds to $\delta = 0$. According to OAS, we should multiply the clock rate by $N^{-\delta/2}$, which is 1 (no reduction). For serial algorithms, OAS tells us to run the entire algorithm on one processor at full clock rate. (This is, in part, the rationale for including a fast thread CPU in Figure 13.)

Choosing between Figure 14A and Figure 14B involves a tradeoff. The example 2D map in Figure 14B could be redistributed across the 3D structure of the PIMS system with an additive cost for the redistribution. Alternatively, the 2D map could be left in place and processed on just a 2D slice of the 3D computer. To be optimal, the 2D slice of the computer should run with a clock rate reduction of $N^{-\delta/2}$ for δ=2/3 or $N^{-1/3}$, which is a higher clock rate and less energy efficiency improvement. The user must weigh the additive cost of redistribution against the multiplicative cost of less parallelism and higher total energy.

Optimization of storage format

Figure 14C expands the vision from Figure 14B to a form that we will later give an architectural realization. Let us assume the computer system is used to store several data sets, with two such data sets illustrated in magenta and green in Figure 14C. Each data set will be stored in devices distributed in a user- or operating system-definable subset of the overall system.

Figure 14C suggests a further innovation related to the representation of data in a storage medium. Data has been stored in “interchange formats” historically. In the past, data has been stored on media that have several attributes that strongly suggest using machine-independent data representations. For example, a disk drive or removable media could be taken to a computer with a different architecture, suggesting encoding formats should be independent of byte order and numerical representation. Data files are also loaded and stored to different memory addresses each time they are accessed by the same computer, arguing against putting pointers or integer offsets into data files. However, the structure in Figure 14C would make it impossible to separate stored data from the computing structure without cooperation of the computing structure. This change in architecture suggests an object store approach that stores data in an opaque internal format and provides access by functions that reformat the data as needed. The scenario in Figure 14B already redistributes data spatially to minimize data movement during computing. The innovation in Figure 14C is to manage the format of stored data to maximize compute efficiency.

There is precedent for this method of data storage in the way we humans store data in our brains. When humans input a pdf file into their brains by reading it, they do not store the bits of the pdf file in their brains, but rather parse and process the document's ideas and store the ideas in a form optimized for making use of the ideas.

Operating system for PIMS

We now have an outline of an operating system for a system behaving according to OAS:

- Data will be stored in the structure of the computer, possibly reformatted, redistributed, and positioned. The method of optimizing storage is discussed below, but once data is stored these factors are used for access.
- To run a function, the operating system evaluates a series of choices and picks the one with lowest cost. The data can be relocated to a position where processing is more efficient, as illustrated in Figure 14A. This incurs an additive cost. The data can alternatively be processed where it is stored, as illustrated in Figure 14B. This has a multiplicative cost in terms of time and energy efficiency if the data is not near enough processors for good throughput.
- On a longer time scale, the operating system will attempt to reformat, redistribute, and position data as illustrated in Figure 14C in order that the data is close to where it needs to be and in the right format for efficient processing.

The conclusion to this section is that there are more issues with the von Neumann architecture than just the fondly named "von Neumann bottleneck" between the processor and memory. We see that integrating processor and memory can reduce and possibly eliminate costs associated with input and output. Also, proper distribution of data in the integrated processor-memory combination can raise the amount of parallelism available for computation. The increase in parallelism enables a lower clock rate and can allow a rise in the power efficiency of the underlying gates. However, there are situations (such as unparallelizable algorithms) where PIMS offers no advantage.

8. Derivation of PIMS tile behavior

This section will show how to derive the behavior of PIMS tiles in Figure 8 for a sparse vector-matrix multiply example problem. The exposition will start with dense vector-matrix multiply, but quickly generalize to a sparse version. The example will be expressed as data movement and generic computation without reference to the underlying data type and operations. Sparse vector-matrix multiply can be applied to data types like bits or floating point variables as long as they have add and multiply operations that create what is known mathematically as a semi-ring. By choosing the proper semi-ring, these tiles can be applied to supercomputing, neural networks, etc.

The diagrams were developed with the help of a spreadsheet. The spreadsheet includes cells near the top where a vector and a matrix can be typed in. When a user types over these cells, the spreadsheet immediately recalculates calculates and displays the vector-matrix product. While the added functionality of the spreadsheet may be minimal the first few times it is used, the calculation is progressively transformed until it unrecognizable. Seeing that the spreadsheet continues to produce the correct result can give reassurance that the transformations were correct. Obviously, a printed version of this document will not have spreadsheet functionality, but the spreadsheet should be available in supplementary material.

PIMS will eventually turn out to be a general graph processing engine, but the exposition will start with a dense vector-matrix multiply in the form of dataflow [Dennis 80] and systolic arrays [Kung 79] (we assume the reader will be familiar with dataflow and systolic array notation). For example, Figure 15 shows a 2D systolic array where the DPU units fill a similar role to the PIMS replication unit.

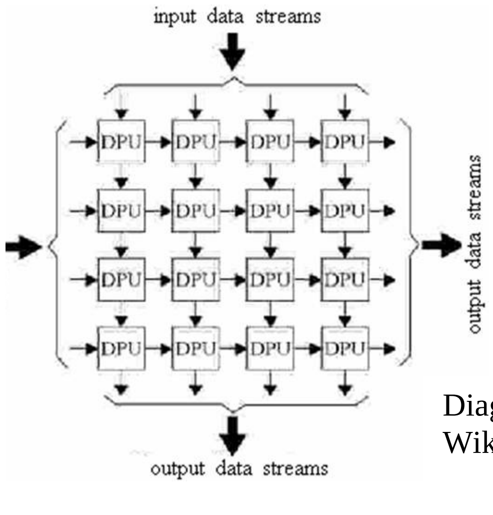
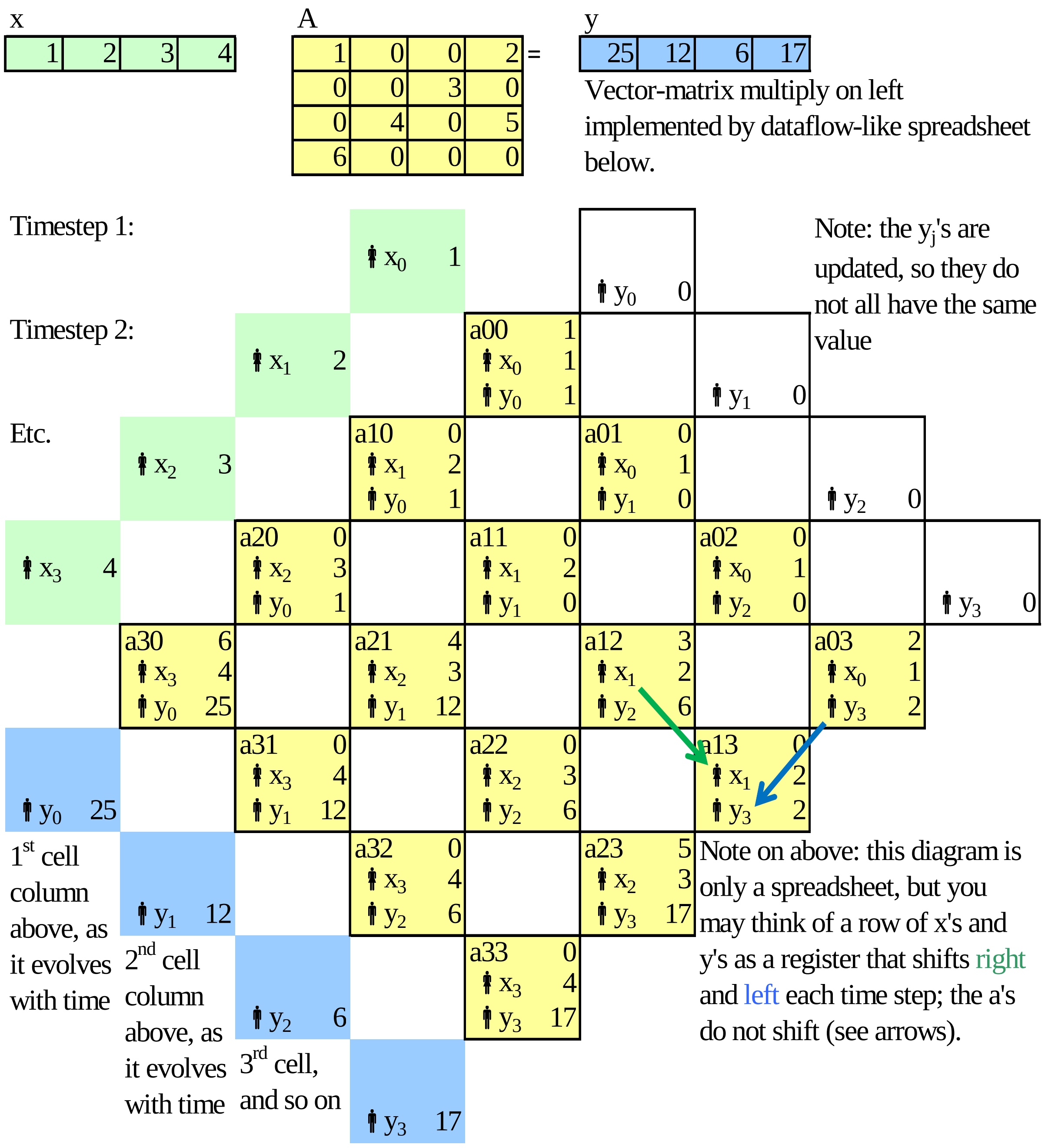


Diagram from Wikipedia

Figure 15: Systolic array

Dense vector-matrix multiply starting point

Our exposition begins with the spreadsheet for a 4×4 vector-matrix multiply in Figure 16, which is similar to Figure 15 rotated clockwise 45°. The figure comprises a functional spreadsheet embedded in the word processor document, where the vector *x* and matrix *A* in the upper left are multiplied through formulae in the spreadsheet cells (but not visible) to create the vector *y* at the center of the top of the diagram.



Vector-matrix multiply on left implemented by dataflow-like spreadsheet below.

Note: the y_j's are updated, so they do not all have the same value

Note on above: this diagram is only a spreadsheet, but you may think of a row of x's and y's as a register that shifts right and left each time step; the a's do not shift (see arrows).

Figure 16: Diagram of the spreadsheet for 4×4 vector-matrix multiplication

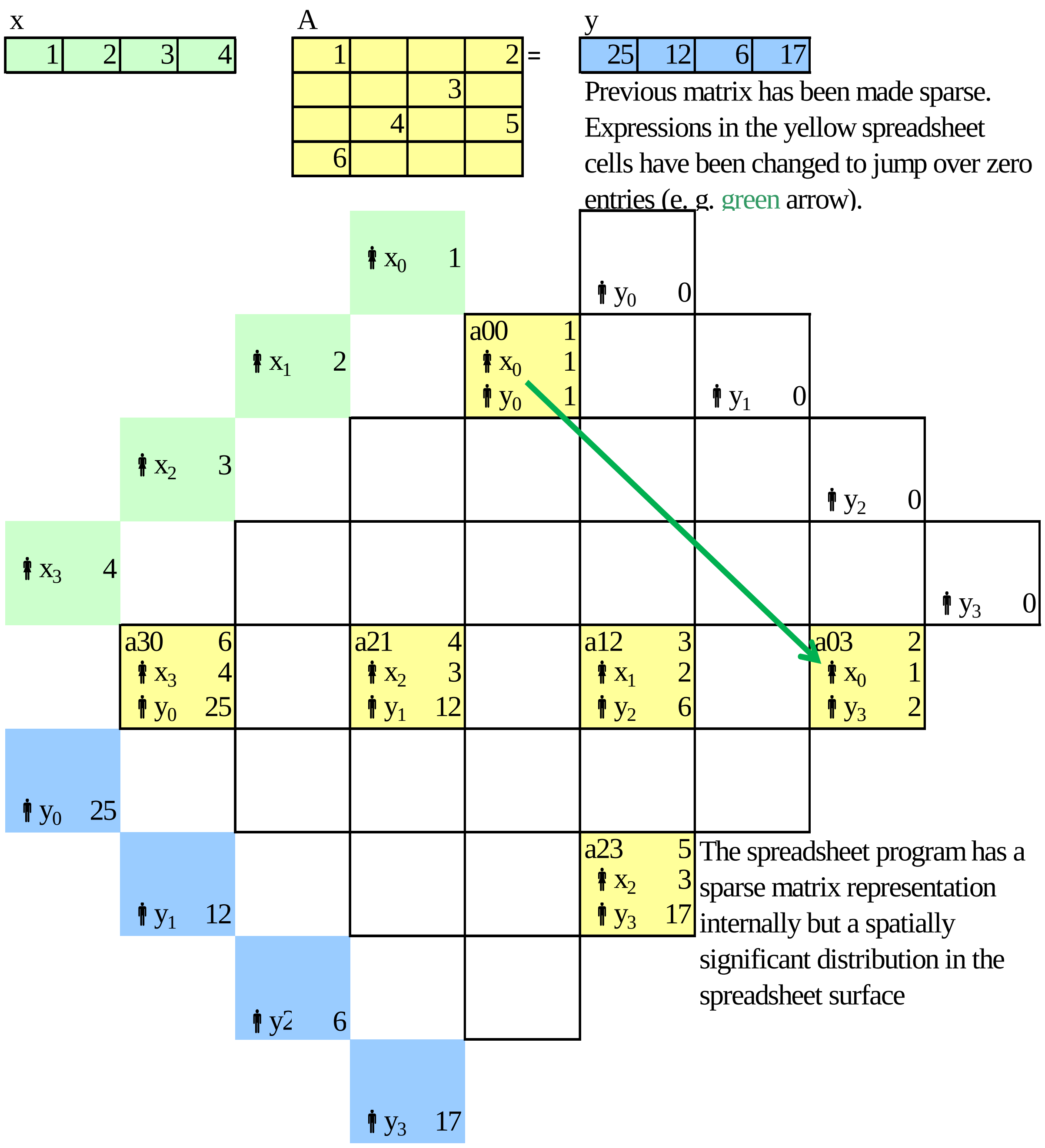
The calculation is performed similarly to a systolic array like Figure 15. Using systolic array notation, input variables x_j and y_k flow down-right or down-left each clock cycle. The *y*'s are initially zero, with each yellow square computing y += $a\ x$, for *y*, *a*, and *x* with whatever subscripts they may have in the cell when the calculation is performed. The variables y_0-y_3 exit to the blue output cells in timesteps 6-9, and are the vector-matrix product of $A\ x$.

Strictly speaking, Figure 16 is just a spreadsheet. The spreadsheet is in the visual form of the original matrix problem, so that it may be a tool for eventually understanding how problems are translated onto the PIMS architecture. The a_{ij}'s in each cell will eventually represent the storage of the matrix in the PIMS memory, with the 2D structure on the page representing data layout in memory. As previously discussed, the adiabatic clocking circuitry reads out the memory an entire row at a time, from top to bottom, so the vertical dimension of the diagram may also be imagined as a time sequence. The x_i's and y_j's will be stored in registers of the PIMS ALUs. For x_i's and y_j's, the diagram must be thought of as a time sequence. In other words, each column represents one ALU at different time steps in the algorithm.

Except for the $x\ A = y$ at the top of the diagram, all the spreadsheet dependencies in the colored cells move downward. This means the computation is a Directed Acyclic Graph (DAG) of arithmetic operations

Sparse vector-matrix multiply

We switch to a sparse matrix in Figure 17. Some entries in the A matrix at the top of the page have been blanked. These entries are permanently designated zero and the calculation has been changed by deleting formulae that multiply or add a permanent zero. This means spreadsheet references that depend on a value in a deleted formula must have their cell references moved upwards (diagonally). For example, the x_0 entry from a_{00} to a_{03} as illustrated by the green arrow. These deletions should not alter the result of the calculation, and the reader will see that the computed y at the top is the same.



Previous matrix has been made sparse. Expressions in the yellow spreadsheet cells have been changed to jump over zero entries (e. g. green arrow).

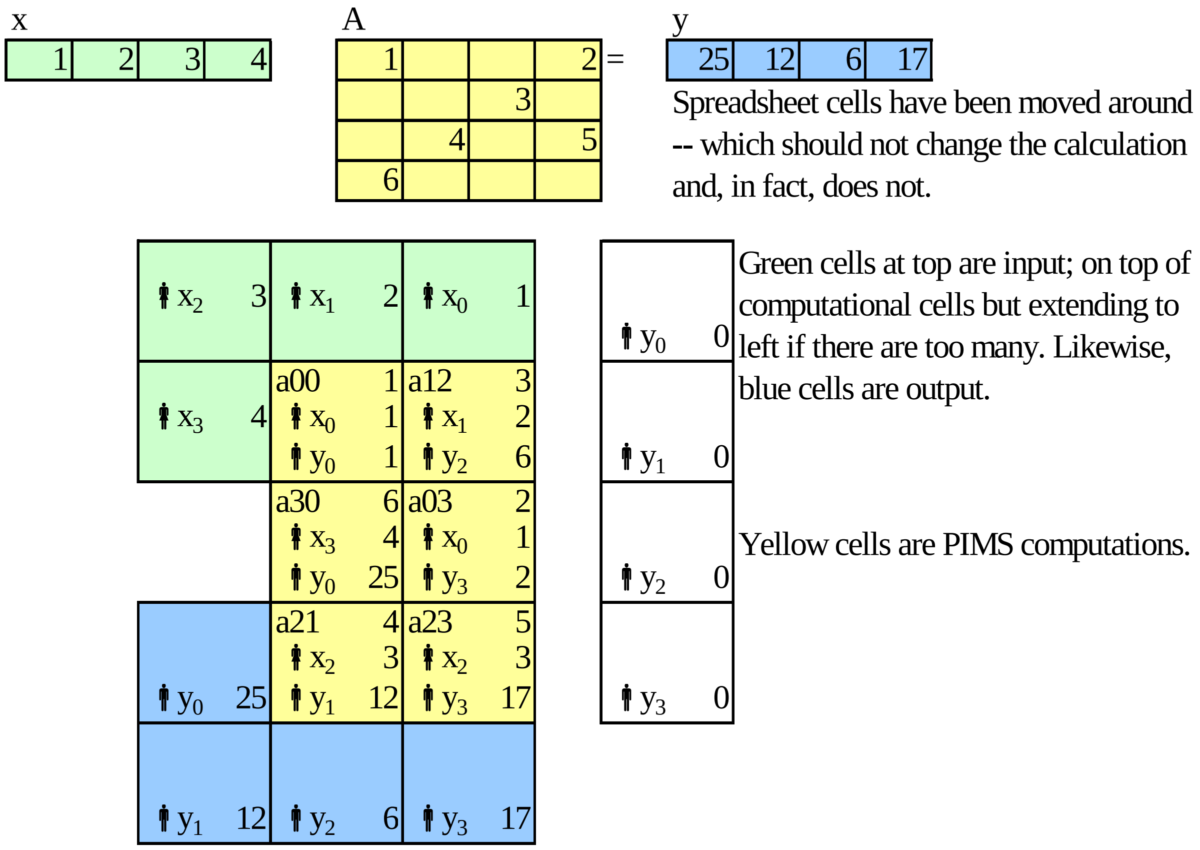
The spreadsheet program has a sparse matrix representation internally but a spatially significant distribution in the spreadsheet surface

Figure 17: Sparse matrix

The spreadsheet is internally maintaining a representation of a sparse matrix through its internal cell references, a fact that will become important shortly.

Transformation to PIMS program

The next step in the exposition involves reordering the calculation to a form that is visually different but in fact represents the same calculation. From the perspective of the spreadsheet that is being used to generate these figures, we have moved cells from Figure 17 to different, previously empty, locations in Figure 18. This should have no effect on the spreadsheet calculation, and the reader will observe that the computed *y* value at the top does not change. Reorganizing the cells in the way the author has chosen destroys the visual tie to a dense matrix, which we suspect some readers will find disturbing. However, this reorganization is key to efficient execution. The ultimate intent is that the PIMS user or programmer would think about matrices in the form of Figure 16 or Figure 17, but that the computer would use an internal graph layout program to choose the more efficient representation in Figure 18. This would be perhaps equivalent to programmers of today's computers thinking about a program in C++ but using processors that actually execute machine code.


x
1 2 3 4
A
1 2
3
4 5
6
=
y
25 12 6 17
Spreadsheet cells have been moved around -- which should not change the calculation and, in fact, does not.
x2 3 x1 2 x0 1
x3 4
a00 1 x0 1 y0 1
a12 3 x1 2 y2 6
a30 6 x3 4 y0 25
a03 2 x0 1 y3 2
a21 4 x2 3 y1 12
a23 5 x2 3 y3 17
y0 25
y1 12 y2 6 y3 17
y0 0 y1 0 y2 0 y3 0
Green cells at top are input; on top of computational cells but extending to left if there are too many. Likewise, blue cells are output.
Yellow cells are PIMS computations.
Figure 18: Reorganization to a PIMS layout

We will discuss the purpose of the reorganization in Figure 18, but a discussion on how the reorganization is performed will be deferred until later. The overall strategy in Figure 18 is to create a sandwich with input (green) at the top, computation (yellow) in the middle, and output (blue) at the bottom. Since time progresses top to bottom, the readout order of the memory will follow input-compute-output, as desired. The reordering must also obey a top to bottom data flow for each internal operation, which can be tricky as we will see.

Input and output

Input and output formats must be consistent so subroutines can interface with each other, but can be arbitrary as long as they are consistent. The author's idea (which is arbitrary but used consistently in this document) is that input and output would reside in the *x* and *y* registers before and after a code block executes, at least to the extent they fit. However, in many cases there will be more input or output than *x* or *y* registers. (This is the case in Figure 18: There are four *x*'s and *y*'s but only two yellow columns.) If there is too much input, the extra inputs would be "shifted in" as the code block executes. If there is too much output, output data will start shifting out early. This behavior is captured by the L-shaped green and blue regions in Figure 18. The inputs and outputs also appear in their proper order, subject to their going around the corner of the L.

PIMS execution

If the spreadsheet dependencies in the calculation (i. e. not including $x A = y$ interface region at the top) all point within 45° of downward (i. e. down, down-left, or down-right), it will be possible to evaluate the diagram on a PIMS. This assertion relies on the following points:

1. We may choose to store the a_{ij}'s in the physical storage of the PIMS any way we like, and we choose to store them with the placement illustrated by Figure 18. This means the energy-efficient adiabatic row access of the memory will send the a_{ij}'s to the ALUs at the proper times without any additional effort.

2. If the *x* and *y* dependencies point within 45° of downward, the PIMS architecture will be able to easily move the data to the proper place between clock cycles with no extra delay. This is because data either moves to a nearest neighbor or does not move at all. Nearest neighbor moves are simple electrically.

3. We do not include dependencies for the initial values or *y*, which are zero for this problem. We assume each ALU can generate a zero on the fly without receiving it from an input link. A different problem might have two inputs.

The spreadsheet in Figure 18 can be used to generate a step-by-step operation chart for the PIMS performing sparse vector-matrix multiplication. Time flows from top to bottom in the chart of Figure 19 in steps equal to one clock cycle or one memory access. Each horizontal row represents the ALUs and their contents at a particular clock cycle. The *a* variable is the output of the memory during that clock cycle and *x* and *y* represent registers. There will be an adder and multiplier discussed later that computes $y \mathrel{+}= a\, x$ as needed for the calculation.

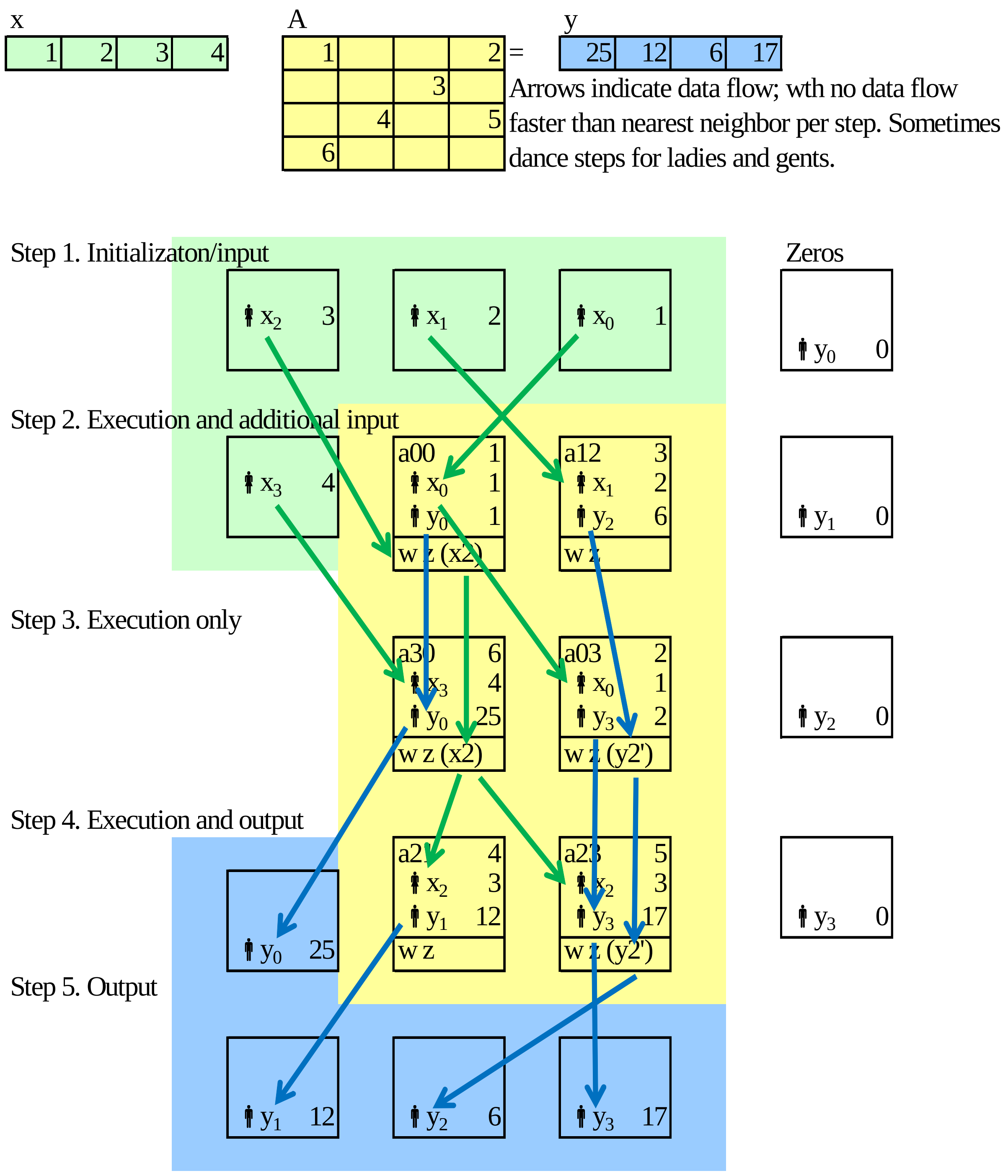

x
1 2 3 4
A
1 2 3 4 5 6
=
y
25 12 6 17
Arrows indicate data flow; wth no data flow faster than nearest neighbor per step. Sometimes dance steps for ladies and gents.
Step 1. Initializaton/input
Step 2. Execution and additional input
Step 3. Execution only
Step 4. Execution and output
Step 5. Output
Zeros
a00
a12
a30
a03
a21
a23
w z (x2)
w z (y2')
w z

Figure 19: PIMS operation chart for vector-matrix multiply

There will also need to be a wait zone register (designated as “w z” due to space limitations on the page). The justification for why wait zones are needed will be given later, but for now let us just say that they are registers that can hold data as it is being shifted to its proper destination tile.

The blue and green arrows represent movement of data values; they are equivalent to the “dance instructions” for the ladies and gents in some descriptions. During each cycle, the memory will send two types of data to an ALU. We have already discussed the memory sending a numeric value to the *a* value in the ALU. However, the memory will also contain an opcode indicating which sequence of data movements apply to the current step. Each cell in Figure 19 sends up to one value to the left and up to one value to the right. Each cell may receive values from left or right, which are stored into registers whose identity has to be specified. The opcode thus contains the encoding of the specific register combination. In other words, the data movement pattern shown by arrows in Figure 19 is determined by the opcodes in each cell.

One of the more complex data pathways will be discussed in a narrative fashion as a check of the reader's understanding. The second cell containing x_2 in step 1 takes a long path before being used. The reader will first observe a green arrow representing x_2 moving to cell a_{00}, its neighbor on the right, and being stored in the wait zone. A straight-down green arrow shows x_2 stays in the wait zone as time passes from step 2 to step 3. In the transition from step 3 to step 4, the value x_2 in the wait zone of a_{30} is transferred to the *x* register of both cells x_{21} and x_{23}. This requires x_2 be copied, with one copy sent to the right.

PIMS compiler and automatic placement

The discussion so far presented the reorganization from Figure 17 to Figure 18 as a *fait accompli*, but we will now discuss how the reorganization is possible through a constructive method.

The needed reorganization is a mapping from the directed graph structure represented by the (hidden) data dependencies in Figure 17 to a compact 2D representation satisfying various constraints. We have already discussed constraints such as matching the specified positions of the input and output cells and the requirement that data flows within 45° of downward. There are other constraints, some of which may be obvious. It is implied that a single register can only hold one value at a time. However, we anticipate that the hardware would only support one data transfer between nearest neighbors in one time step. In other words, it would not be permissible to have more than one arrow between the same cells in the same direction on the same time step. For example, a cell may not send both its *x* and *y* values to the left in the same time step.

There are software algorithms for the required reorganization, but the algorithms have limitations that need to be managed. The general algorithm class is used in such tools as graph layout for publishing diagrams in documents and place-and-route tools for integrated circuit layout. The graph layout algorithms are based on a minimum cost flow function called network simplex [Ellson 04].

The sparse matrix graph in Figure 17 was entered into the GraphViz tool, with the output shown in Figure 20. Based on the discussion of the input and output cells in Figure 19, the author manually entered a constraint that these cells would be in an L-shaped configuration (as shown). The GraphViz tool then placed the remaining yellow nodes automatically. The reader will see that this resulted in a placement with both similarities and differences to the placement in Figure 18. If the a_{12} node is imagined to be moved up one rank, the graphs in Figure 18 and Figure 21 will be the same except for horizontal positions. GraphViz is a layout tool with an aesthetic objective as opposed to meeting the PIMS constraints (which would have been entirely unknown to the developers of GraphViz). Both the difference in horizontal placement and the position of the a_{12} node are arguably more aesthetic.

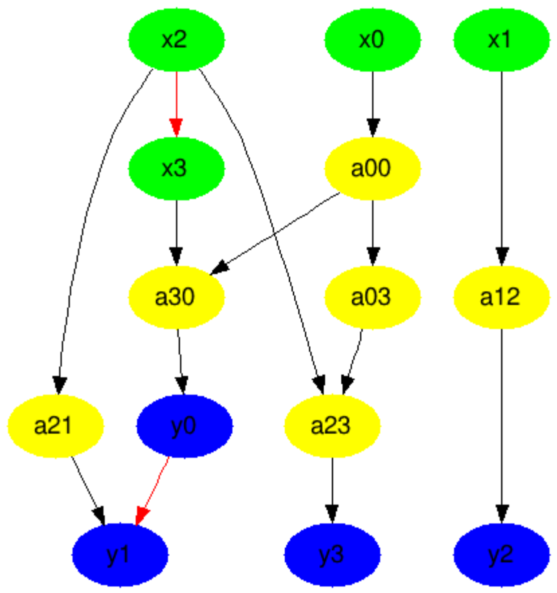


Figure 20: GraphViz applied to the sparse matrix example

The author’s plan would be to develop a special reorganization tool for PIMS, much like each microprocessor type today has a compiler.

The underlying placement algorithms have limits in quality of the result and run time. It is known that the underlying network simplex algorithm is NP-hard, but run time tends not to be an issue because a “pretty good” solution is all that is needed. Furthermore, the optimal solution may not be as compact as one might like. The most compact representation of certain graphs may require an extremely tall graph or a large number of a wait zone registers. However, it is reasonably easy to show that any graph can be embedded into the form of Figure 19 if one is willing to have unused cells.

The benefit of the reorganization should be obvious but will be restated. A computation with n operations could fill a matrix as large as some moderate fraction of n^2, such as ½ n^2. With the compression resulting from the reorganization suggested, the total number of PIMS cells could be reduced to n (where the number of cells refers to the product of the number of physical cells required and the number of steps) in the best case. In reality, the optimal placement may not meet this best-case limit, and a heuristic algorithm may not find the optimal placement. However, the benefit of the reorganization is a reduction in the resources required for execution on a PIMS from $O(n^2)$ to $O(n)$.

Instruction set architecture development

An example Instruction Set Architecture (ISA) will be developed for the specific example problem. This will only be an example of an engineering process that will need to be repeated in the future, because a production computer would either have to be more general or more highly tuned to specific problem class.

The first step in the ISA development will be to reorganize the example into fixed, large-scale computer structure plus instructions that customize behavior for a particular problem. After some thinking, it appears that the "instructions" can be represented as tiles in the form of Figure 21. These tiles would be defined such that none of the movements need look further than the boundary of the tile. The job of the computer structure interconnecting the tiles will be to pass data between cells as the adiabatic clocking accesses rows. This implies activating one row at a time in time sequence, (1) putting the contents of memory in the *a* register, (2) moving y_{out} to y_{in} and wz_{out} to wz_{in} between cycles, and (3) doing the nearest neighbor swaps r_{out} to l_{in} and l_{out} to r_{in} between cycles.

	yin	wzin	
lin	a		rin
	x		
	y		
lout	wz		rout
	yout	wzout	

Legend:
y_{in}/y_{out} = *y* register input/output
wz_{in}/wz_{out} = wait zone input/output
l_{in}/l_{out} = left input/output
r_{in}/r_{out} = right input/output

Figure 21: Instruction format

It will now be possible to build a spreadsheet emulator for PIMS using the ISA, as illustrated in Figure 22. The spreadsheet is very similar to those shown previously, but some spreadsheet dependencies have been changed into sequences of dependencies to simulate the specific data movement required for tiles like Figure 21. This has been accomplished without changing the behavior of the example. For example, the spreadsheet formulae in rows 3-6 of Figure 22 have been changed so they do not go outside the figure. Also, the formulae in rows 1 and 2 are the data movements of the computer structure and all go outside the figure. Figure 22 is messy and included as an example of the required development process.

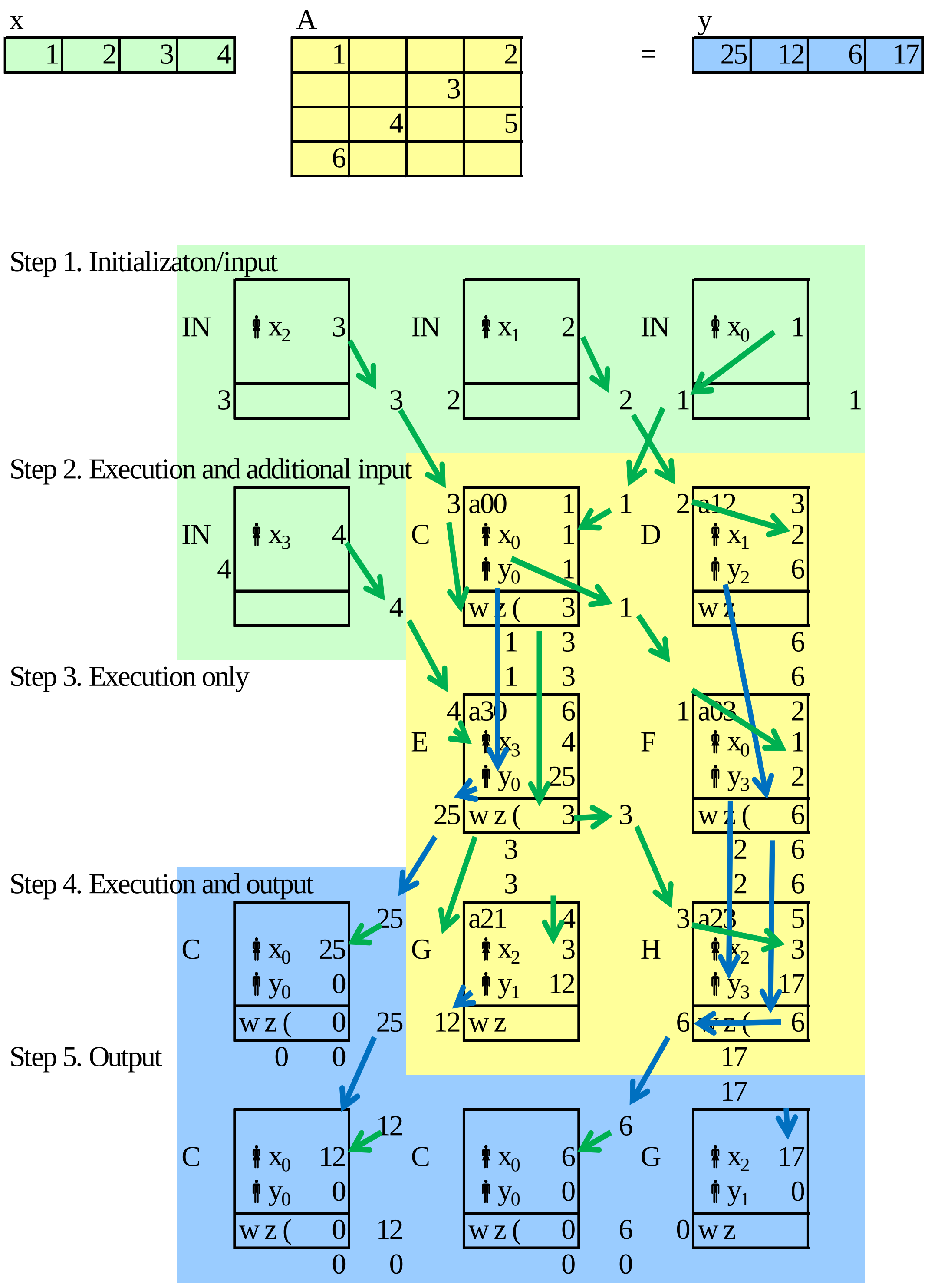

x
A
y
=
Step 1. Initializaton/input
Step 2. Execution and additional input
Step 3. Execution only
Step 4. Execution and output
Step 5. Output
IN
C
D
E
F
G
H
a00
a12
a30
a03
a21
a23
w z (

Figure 22: ISA-based PIMS emulator

The next step is to develop a comprehensive list of instructions, merging them into compatible classes. The worksheet for this is shown in Figure 23, with the author's notes from the merging of compatible cells. The author initially created a list of different cell types by examining Figure 22 left-to-right, top-to-bottom. These cells are labeled A-J in the second column of Figure 23, and to some extent in the cells of Figure 22. The columns of the table represent outputs of each instruction, with the values below indicating the register supplying the output value.

opcode		x	y	wz	lout	xout	yout	wzou	rout	
	A	init							x	now IN
	B	init			x					
1	C	rin	0+a*x	lin			y	wz	x	
2	D	lin	0+a*x					y		
3	E	lin	yin+a*x	wzin	y				wz	
4	F	lin	0+a*x	wzin			y	wz		
5	G	yin	0+a*x		y					
6	H	lin	yin+a*x	wzin	wz		y			
	I		rin							now c
	J		yin							now g
0	IN	init			x				x	

Figure 23: Worksheet for developing instructions

The author noted that the input instructions A and B were compatible and could be merged to create cell IN at the bottom.

Also, the behavior of instructions C and G “sort of” included the behavior of the output instructions I and J. By “sort of,” we mean instructions C and G transfer data from the correct input and in the correct direction, but they transfer it to the *x* register instead of *y*. Upon thinking about a future computer design, that looks like a better choice anyway. So we have replaced the output cells by C and G, dropping I and J from consideration.

The author then developed an opcode numbering for the remaining 7 instructions, which appears in the first column.

The resulting 7 instructions can be drawn graphically as shown in Figure 24.

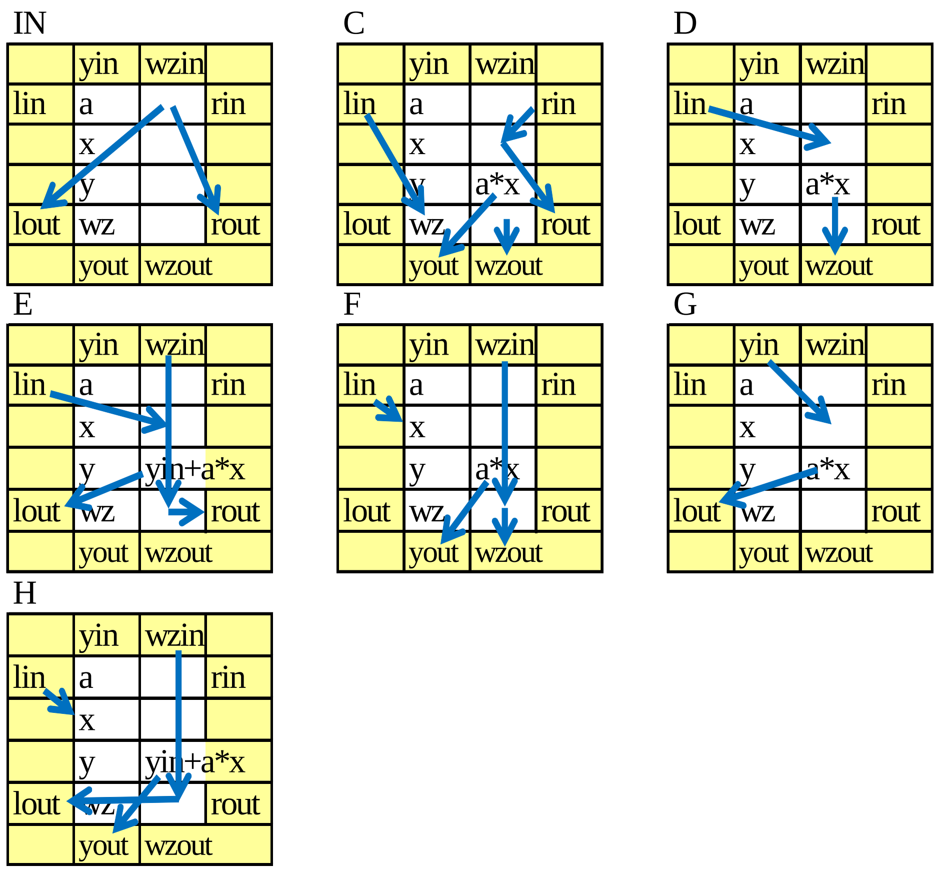


Figure 24: PIMS instruction set for example

With this instruction set, the PIMS memory for executing the program appears in Figure 25. In this figure, the *x*'s and *a*'s represent input values and the codes IN and C-H represent the program.

Test program memory layout

x_2	IN	x_1	IN	x_0	IN
x_3	IN	a_{00}	C	a_{12}	D
		a_{30}	E	a_{03}	F
y_0	C	a_{21}	G	a_{23}	H
y_1	C	y_2	C	y_3	G

Result appears in x registers

Figure 25: PIMS test program

ALU logic design

From Figure 23, we have information sufficient to derive the register-transfer level logic diagram in Figure 26 for each ALU. This logic diagram has been expressed in the layout form of Figure 21, notably the positions of the input and output connectors. The author created the diagram by drawing logic feeding each output that can generate the values required by all the tiles with a suitable choice of control wire settings.

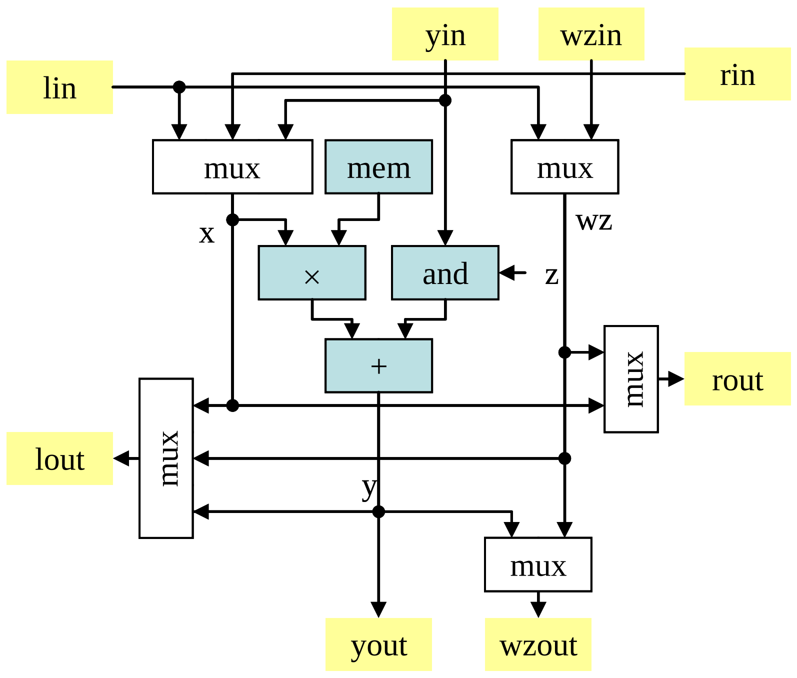


Figure 26: Logic diagram of replication unit

Notes on the logic diagram:

1. The logic diagram does not include any latches. The computer structure would provide latches.
2. The box "mem" is memory output. The example does not include writing to memory.
3. The box "and" would be capable of zeroing the y_{in} signal, allowing differentiation between the calculation of $y = a \times x$ and $y = y_{in} + a \times x$. z would be a control signal.

9. Control system

There have been two previous discussions related to the control system for PIMS, which will be combined here into a scheme for a general computer system. It was disclosed in the context of Figure 11 that a control system should access the PIMS memory sequentially from the top row to the bottom row in a repeating sequence – except that this pattern would be extended later. However, Figure 14 shows the overall memory of a PIMS computer being devoted to several tasks at once. We also need one additional function to allow PIMS to become fully general purpose, for which we will introduce a "conditional subroutine call." The resolution to these three issues will be a two-level structure. One or more PIMS replication units could be connected to form a logical memory bank with straightforward access patterns such as those discussed in the context of Figure 11. However, an overall system could comprise multiple tasks that execute independently. A logical memory bank will then be extended with a conditional subroutine call capable of launching a new task in the overall system.

The behavior of a simple control system is illustrated in Figure 27. The figure illustrates three logical PIMS banks with data/program in the storage array, ALUs, and the control unit.

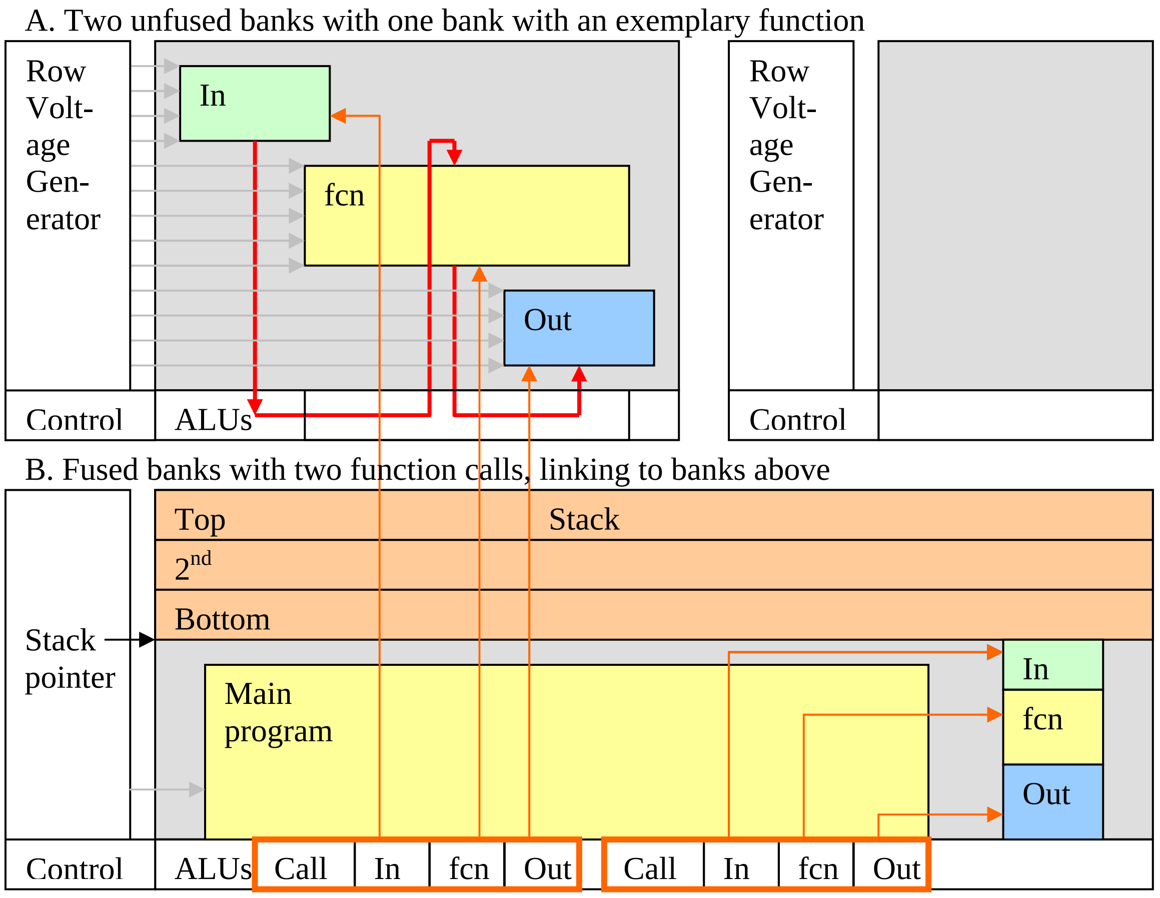

A. Two unfused banks with one bank with an exemplary function
Row Volt- age Gen- erator
In
fcn
Out
Control
ALUs
Row Volt- age Gen- erator
Control
B. Fused banks with two function calls, linking to banks above
Top
Stack
2nd
Bottom
Stack pointer
Main program
In
fcn
Out
Control
ALUs
Call
In
fcn
Out
Call
In
fcn
Out

Figure 27: Control system and subroutines

Interbank architecture

We designate a PIMS system to be comprised of physical banks (where the term “bank” comes from the term “memory bank” and corresponds to a PIMS replication unit discussed previously) that can be dynamically fused into groups called logical banks. However, we will often call any of these just “banks.” All banks are thus behaviorally identical, but vary in data and computing capacity. Of course, PIMS holds non-volatile information, so data stored physical banks stays with the physical bank as the fusing pattern changes. Figure 27 illustrates three banks, meaning each may be a single physical bank or the fusion of several. This approach is useful architecturally, but is also driven by the engineering necessity to build memories out of physical banks of about 1000×1000 bits (in 2D) to avoid excessively long row and column lines. Physical banks may also operate independently and communicate, as illustrated by orange lines in Figure 27 bridging between banks (whose purpose will be described below).

Function layout and execution

Figure 27A illustrates the bank’s storage region with a gray background but containing activity-specific data in colored rectangular regions. The regions correspond in function and color to the data illustrated in Figure 14 and Figure 19, namely In (input), fcn (processing), and Out (output). Like Figure 19, the objective of the control system is to apply input to a function and collect the output, but the control system will also move (or stream) the input and output data if the source or destination region is not in the right place to begin with. So the control system will actually have three temporally interleaved activities:

1. Read rows of input data into the ALUs and then shift the data between ALUs so it is aligned with the function area.

2. Read, write, or read-and-write-back the function data, executing the function defined by the tiles on the data.

3. Shifting output data so it is aligned underneath the output data area, and then writing it into the output data area.

Since data may be in rectangular regions of different sizes and shapes, the control system is expected to have several sets of counters and registers that jump between the three functions above in a potentially irregular sequence.

A command to the control system will include parameters defining the three rectangles for In, fcn, and Out data. There may be other parameters as well. We will later on use a control unit command with only In and Out rectangles, which will move the input data to the output location without changing the data. This variant will be used to create a random access memory; more on this later.

A second PIMS bank is illustrated in Figure 27B in the process of executing two subroutine calls. In the illustration, a row of the main program is being accessed. The row illustrated contains two instruction tiles of a new type named “call” (which is actually shorthand for “conditional call”). In the interface being defined here, each call tile is expected to be part of a larger group (outlined in orange in the figure) followed on its right by three sets of tiles that compute rectangles for In, fcn, and Out. These tiles perform what is traditionally called an address calculation; they produce the position of the rectangle that will hold the data as opposed to the data itself, the positions comprising row and column numbers. The calls in the example command the control system to execute two functions, one whose data is in Figure 27A and the other in Figure 27B.

PIMS subroutine execution process

In addition to executing instruction tiles such as those in Figure 24, the hardware looks for instruction tiles representing the conditional subroutine call. If found, the other tiles are executed, followed by the much more elaborate subroutine processing described below.

The conditional subroutine call behaves as follows: "if (*x* != 0) *y* = *fcn*(*In*, *Out*)," where the subroutine return value *y* would be obtained from a consistent location, such as the *y* register computed by the lower-right cell of *fcn* and transferred to the *y* register of the calling tile. In more detail, if the *x* register in the conditional subroutine call tile is zero, the call is not performed. Otherwise,

1. The state of the control system is pushed onto the stack, including the three rectangles above plus information about how far along the control system has gotten in moving the data.

2. The control system starts execution of the subroutine with the parameters In, fcn, and Out.

3. When the control system reaches the end of the execution of the subroutine, the state of the control system is popped off the stack and execution continues. In addition, the value in the *y* register of the lower right cell of the subroutine is moved to the *y* register of the cell that initiated the call.

If there are multiple calls in a row, it would be possible to use some sort of scheduling strategy to run the subroutines a series-parallel arrangement, including concurrent execution on multiple effective banks. If the calls are expected to execute quickly, the hardware could (a) dispatch calls to other banks, with banks defined as they are fused at the moment of the call, and (b) sequentially execute call within a bank. In situations involving a lot of resources, it may be appropriate to interrupt PIMS execution to run a scheduling algorithm that can additionally alter the grouping pattern of the banks.

General execution

It is important for the objectives of this document to have a strategy for execution of “legacy” code and languages. Our plan is to find a way to compile and run code in modern languages (e. g. C) natively on a PIMS (albeit not necessarily efficiently), yet also enable a skilled programmer to write new code that can access the higher power efficiency available through the architecture. This could evolve into the well-known process or adding an accelerator to standard processor where the accelerator includes a low-level subroutine package. We will create a sufficient albeit non-optimized solution to this issue through a “tail recursion” enhancement to the conditional call above plus further clarification of the two operand control system command.

Loops and conditionals are possible using the method disclosed above, yet loops require a minor hardware adjustment to avoid consuming an unnecessarily large amount of stack. “Tail recursion” can be the basis of a strategy to implement loops on PIMS, which is described with the help of Figure 28. The left side of Figure 28 has blue code that shows a loop in C, where the right side rewrites the loop as a subroutine. The last line in the subroutine conditionally recurses to repeat the loop, and the reader should see that they have the same behavior. Furthermore, the blue code on the right can be implemented with the conditional call discussed above. Unfortunately, implementation of the conditional call above would require a stack depth equal to the number of iterations through the loop.

```
main() {
    do {
        // loop body
    } while (/* continue? */);
    if (c)
        /* true code */
    else
        /* false code */
}
```

```
main() {
    loop()
    if (c) true_code();
    else false_code();
}

loop() {
    // loop body
    int x = /* continue? */
    if (x != 0) loop();
}

true_code() { /* true code */ }

false_code() { /* false code */ }
```

Figure 28: Loop strategy for PIMS

Hardware support for tail recursion [Tail YY] is recommended and straightforward: The control system merely suppresses pushing state to the stack for the last subroutine call on the last line of a function. If this suppression were not implemented, the loop would end with as many immediately sequential subroutine returns as there are iterations in the loop. The structure of the tail recursion guarantees that all but the last subroutine return simply jumps to another return. With the suppression, all these returns are replaced by just the final return – along with just one push and pop of the stack. In fact, the subroutine call has essentially turned into a jump instruction.

Conditionals could be performed by creating two subroutines, as illustrated in red in Figure 28. One subroutine includes the functions in the true branch and the other for the code in the false branch. The main program conditionally calls the subroutine for the true branch, flips the condition, and conditionally calls the subroutine for the false branch.

The computing model described above is nearly sufficient to the target of a modern portable compiler; it is only missing access to memory. However, random access memory can be emulated with the two-operand system controller command described above. The compiler would designate some region in the PIMS storage to be an *n*-word, contiguous, memory for a program. The rectangular area containing the *k*'th word could be computed by some formula, such as shifting the rectangle *k* addresses in some direction. To access the *k*'th word of memory, a program would compute this rectangular area and then execute a control system command with this rectangle and another rectangle that may be the data field in the next row. The system controller would then move the memory contents into the next row of the program, where it can be used. Likewise for writing.

The strategy for a general programming language would be as follows. A compiler turns a C program into a parse tree, which is a graph. The nodes of the parse tree include arithmetic expressions, loops, conditionals, subroutine calls, and memory access. The discussion above has shown PIMS equivalents for all of these. The idea would be to feed the compiler's parse tree into a layout program such as Figure 20, thus turning each C subroutine into a one or more functions like that illustrated in Figure 22. In principle, the process just described could be used to port an operating system onto a PIMS and create a stand alone system. It would also be possible to have the operating system and main programs run on the fast-thread CPU illustrated in Figure 13, using PIMS for computation-intensive lower level computations and accessed in the same memory space as the fast-thread CPU. In fact, a continuum of processing speeds are possible. This has been a very abstract discussion, and it is left as an exercise for the reader to fill in details (!).

10. Sparse matrices for neurons and meshes

Sparse matrices are key to both neural computing and supercomputer algorithms, yet the sparse matrices must apply to the right number system for each application (the correct semi-ring). Supercomputer algorithms include the Finite Element Method (FEM) and graph algorithms. The most compute-intensive activity in any of these is a sparse or dense vector-matrix multiplication.

In an Artificial Neural Network (ANN), it is the multiplication of a vector representing neural axon output signals times a sparse matrix representing synapse weights. In current Deep Learning implementations of ANNs, the underlying arithmetic can be 8 or 16 bit integers or single precision floating point. However, multiplication by both a matrix and its transpose are required.

Finite element methods on supercomputers create an irregular mesh around a structure to be simulated, such as a turbine blade. A compute-intensive step for simulating with meshes is the multiplication of a vector representing, say, force, at each point on the by a sparse matrix representing interconnections between mesh points. FEM simulations also require operation on both a matrix and its transpose but tend to use double precision floating point arithmetic.

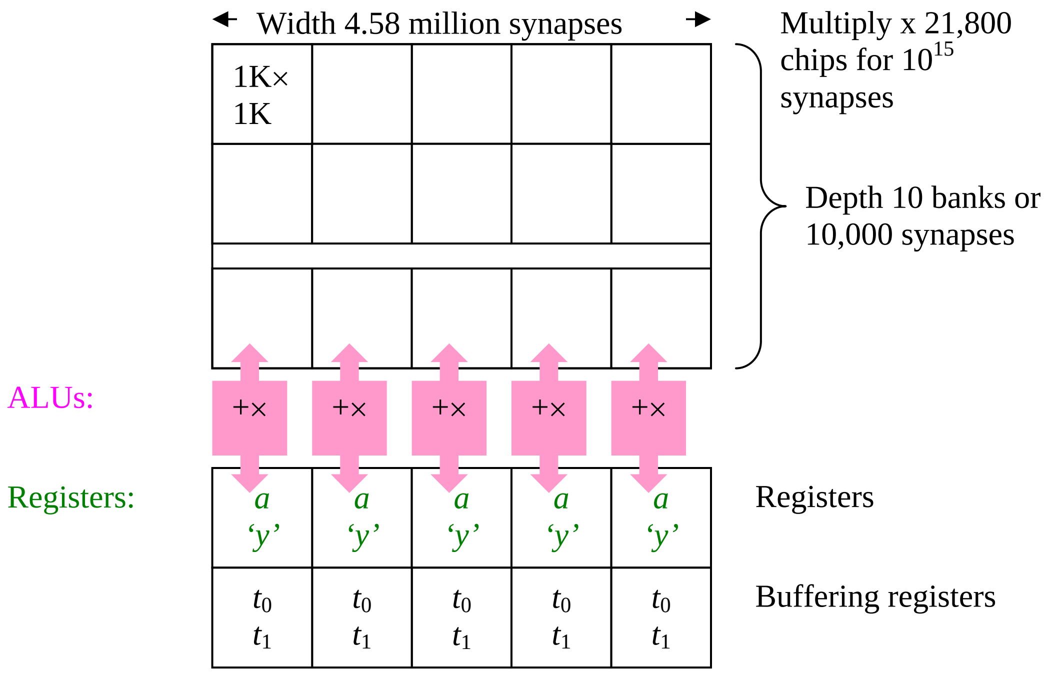
Graphs are often represented by matrices with nonzero elements for graph edges that exist and zeros where there are no edges. Graph algorithms can be represented by sparse matrices like ANNs or FEMs, but where the arithmetic may be non-existent (i. e. the sparsity pattern is the only data).

Power analysis

Let us make an argument that the design shown in Figure 13 and Figure 26 could be power efficient in the short term and progressively more power efficient in the future. Let us identify technologies that have been demonstrated, albeit not in combination. This will yield an optimistic but not impossible result and with some guidance for each sub technology. Let us do this for an ANN system of human brain proportions of 10^{11} neurons, 10^{15} synapses, and a 20 Hz update (i. e. brain wave cycle) rate. Let us initially assume the system is a collection of chips like Figure 13; whether these chips are themselves stacked will be considered later.

A rough sketch of such a system is shown in Figure 8. Let us assume an equivalent storage density to a state-of-the-art Flash memory chip containing 64 GBytes or 512 Gbits (e. g. Micron MT29F512G08CUCAB [MT29F512G]). Since a synapse is represented by 12 bits, this implies $64\times2^{30}\times8/12 = 4.6\times10^{10}$ synapses per chip, or 10^{15} / 4.5×10^{10} = 21,800 chips will be required to implement 10^{15} synapses. If a memory or storage bank is 1000 × 1000 bits, each PIMS chip could contain 55,000 × 10 one-megabit banks, representing 4.58×10^{6} × 10,000 synapses. The structure just described would have the correct ratio of 10,000 synapses to each neuron.

PIMS chip equivalent to 64GByte Flash:



Width 4.58 million synapses
Multiply x 21,800 chips for 10^{15} synapses
1K× 1K
Depth 10 banks or 10,000 synapses
ALUs:
+×
Registers:
a 'y'
Registers
t_0 t_1
Buffering registers

Figure 29: One of 21,800 chips in a brain-sized structure

The assessment approach will be to separately evaluate memory and logic options, followed by a cross-product table for systems. In more detail: While the memory and logic will be tightly integrated, their power efficiency can be evaluated separately. A system in our analysis will be specified by three parameters:

1. $E_{memoryBIT}$, the energy for the memory to access each bit in the whole-row-at-a-time mode
2. N, the number of bits in a synapse, including 4 bits for pointers
3. $E_{logicALUP}$, the energy to process a synapse (for all the bits in the synapse)

The energy to evaluate a synapse will be

$E_{synapse} = N\, E_{memoryBIT} + E_{logicALUOP}$

We will analyze two memory options and four logic options. These will create eight combinations. We will also analyze a commercial GPU, which yields a single result with no breakdown by logic and memory.

Memory Access Energy

We will need to estimate energy access energy, which we will do for both conventional DRAM and an extrapolation of an adiabatic research device [Karakiewicz 12].

The most energy-efficient method of accessing large amounts of memory in a production chip occurs during DRAM refresh. A DRAM such as the 4 GBit Micron MT41K256M16 (we will consider the 1066 MHz version) [DDR3L] refreshes its entire contents in 8192 refresh cycles, or $2^{32}/2^{13} = 2^{19}$ = 512K bits each refresh cycle.

Equation (22) on page 15 of [TN-41-01] is:

$\mathrm{Pds(REF)} = (I_{DD}5 - I_{DD}3N) \times V_{DD}$

In this equation Pds(REF) is the power of the subcomponent REF, where REF stands for "refresh." For more details, see [TN-41-01].

From the datasheet for the part above [DDR3L], $I_{DD}5B$ = 205 mA, $I_{DD}3N$ = 68 mA, and V_{DD} = 1.35v (table 19, p. 41). This yields 185 mw power during refresh. The time for each refresh cycle is 139 clocks (table 8, p. 30) at a clock rate of 1066 MHz, or 130 ns.

The refresh energy per bit is therefore

$E_{DRAMBIT} = 185\ \mathrm{mw} \times 130\ \mathrm{ns}/2^{19}$ joules/bit = 46 fJ/bit

The second option is a unique adiabatic DRAM described in [Karakiewicz 12]. The paper reports that when adiabatic charge recycling is turned on, energy efficiency improves by 85×, which implies about 98.8% of the energy driving rows and columns is recovered. Unfortunately, this 256-row device is used in logic mode where on average 128 rows are accessed in parallel, with the number of "1" bits added in analog. We will now make a wild assumption. Most of the power in this research memory is in the row drive, as opposed to the logic. So we will assume/speculate that we could achieve the same power efficiency driving 128 memory banks one row at a time. In this case, the adiabatic memory approach would yield extremely good results.

The research chip in [Karakiewicz 12] achieves 1.1×10^{12} ops/second/milliwatt. If the ops can be separated across banks and the device operated purely as a memory, energy per bit will be

$E_{TMACSBIT} = .001 / 1.1 \times 10^{12} = .91$ fJ/bit

The nVidia GTX 750 Ti is a state-of-the-art consumer GPU today. Memory bandwidth is limited to 86.4 GBytes/sec. The system consumes 60W, yielding 86.8 pJ/bit. If a synapse is 12 bits, the energy requirement will be 1.04 nJ/synapse to load the synapse value from memory to the processor. This subsystem is memory bandwidth limited in this application, so we will assume the computing energy is zero.

Compute Energy

The publication [Nikonov 13] contains energy projections for Beyond CMOS devices, including 32 bit adders. We will consider two devices: the "HomJTFET," which is the device with lowest energy and “CMOS HP,” which is the common device in microprocessors. The energies per operation are:

$E_{\text{TFET-32Add}}$ = .15 fJ/add (figure 47, p. 65),
$E_{\text{HP-32Add}}$ = 2.5 fJ/add (figure 47, p. 65),

for a 32-bit adder. Dividing by 32, this is

$E_{\text{TFET-FA}}$ = .15 fJ/32 = 4.7 aJ
$E_{\text{HP-FA}}$ = 2.5 fJ/32 = 78 aJ

Each synapse evaluation includes an 8 × 8 multiply, a 16-bit add, and what we will estimate as a tripling of energy for control and bit transfer. Let us assume an *N*-bit multiplier consumes $1.2N^2$ times the energy of a full adder. The 20% addon is due to the need for gated full adders. This would lead to

$E_{\text{TFET-mul}}$ = (64 + 20%) $E_{\text{TFET-FA}}$ = 77 $E_{\text{TFET-FA}}$ = .36 fJ
$E_{\text{HP-mul}}$ = (64 + 20%) $E_{\text{HP-FA}}$ = 77 $E_{\text{TFET-FA}}$ = 6 fJ

However, we will also include an estimate for 21-bit multipliers to be closer to single precision floating point

$E_{\text{TFET-mul-21}}$ = (21^2 + 20%) $E_{\text{TFET-FA}}$ = 529 $E_{\text{TFET-FA}}$ = 2.4 fJ
$E_{\text{HP-mul-21}}$ = (21^2 + 20%) $E_{\text{HP-FA}}$ = 629 $E_{\text{TFET-FA}}$ = 41 fJ

Let us assume the adder is 16 bits

$E_{\text{TFET-add}}$ = 16 $E_{\text{TFET-FA}}$ = 75 aJ
$E_{\text{HP-add}}$ = 16 $E_{\text{HP-FA}}$ = 1.25 fJ

This will lead to energies per synapse evaluation in an ALU

$E_{\text{TFET-ALUOP}}$ = 3 ($E_{\text{TFET-mul}}$ + $E_{\text{TFET-add}}$) = 1.3 fJ
$E_{\text{HP-ALUOP}}$ = 3 ($E_{\text{HP-mul}}$ + $E_{\text{TFET-add}}$) = 22 fJ
$E_{\text{TFET-ALUOP-21}}$ = 3 ($E_{\text{TFET-mul-21}}$ + $E_{\text{TFET-add}}$) = 7.7 fJ
$E_{\text{HP-ALUOP-21}}$ = 3 ($E_{\text{HP-mul-21}}$ + $E_{\text{TFET-add}}$) = 128 fJ

System Energy

If extended to 10^{11} neurons and 10^{15} synapses updating at 20 full cycles per second, we construct Table 3 showing overall system energy in orange for the pertinent combinations of memory and logic options. Four of the rows list a logic energy in blue and two of the columns list an memory energy in red. Each row-column intersection four constituent energies (see legend) and the system energy in kilowatts. The right column for GPUs represents a memory-bound situation and is modeled by zero additional energy for computation.

Table 3: Power/energy for various combinations of technologies

Memory Logic type	GTX 750 Ti 0.1 nj/bit	DRAM 46.0 fj/bit	Adiabatic Mem 0.9 fj/bit
TFET 1.3 fj/synapse 12 bits needed	1.0 nj 0.0 j 1.0 nj 20.8 mw	552.0 fj 1.3 fj 553.3 fj 11.1 kw	10.9 fj 1.3 fj 12.2 fj 244.3 w
CMOS HP 21.8 fj/synapse 12 bits needed	1.0 nj 0.0 j 1.0 nj 20.8 mw	552.0 fj 21.8 fj 573.7 fj 11.5 kw	10.9 fj 21.8 fj 32.7 fj 653.2 w
TFET 21 bits 7.7 fj/synapse 25 bits needed	2.2 nj 0.0 j 2.2 nj 43.4 mw	1150.0 fj 7.7 fj 1157.6 fj 23.2 kw	22.7 fj 7.7 fj 30.4 fj 607.9 w
CMOS HP 21 bits 127.8 fj/synapse 25 bits needed	2.2 nj 0.0 j 2.2 nj 43.4 mw	1150.0 fj 127.8 fj 1277.7 fj 25.6 kw	22.7 fj 127.8 fj 150.5 fj 3010.2 w
Line 1: Femto joules to access memory for one synapse Line 2: Femto joules logic energy to act on one synapse Line 3: Sum of previous two lines Line 4: System energy (watts, kilowatts, megawatts)			

A conspicuous result of the Table 3 is that all the PIMS approaches beat a contemporary GPU by 2000× or more.

A second result is that DRAM energy dominates logic, when DRAM is used. The energy to access a synapse with DRAM is much higher than the energy to process the synapse. Specifically, for the four scenarios, the memory access energy is 9-425 times larger than the logic energy. This means the logic is "essentially free" in this scenario. If this situation were likely to persist, the wise architect would add complexity to the logic.

If our wild assumption that the adiabatic DRAM is feasible, the TMACS row achieves a substantial further improvement.

However, the extremely low access energy demonstrated in [Karakiewicz 12] raises the possibility that the logic in Figure 26 is actually overbuilt. GPU and PIMS are subject to different design constraints. GPUs have always had an external memory with an access energy limited by the DDR or DDR-class bus between chips. It would be silly for a GPU architect to reduce GPU energy below that of the memory feeding it. It would be more sensible for the GPU architect to add additional GPU features even if the features were marginally useful. This is presumably the reason for texture maps, double precision, etc. While the thought process in the sentences above applies to PIMS, the memory access energy is much less and as a result the amount of logic that will balance the memory access energy will be much less. The logic illustrated in Figure 26 might benefit from adiabatic implementation (see next section) which has not been considered in the power calculation.

With the adiabatic memory (as wild a speculation as it is), the system is more power efficient that a modern GPU by around a factor of 80,000.

11. Conclusions

We devised an approach to increasing computer performance beyond the limits of Moore’s Law and the microprocessor. The objective is to minimize a mix of energy cost and the cost to build the computer in the first place. The design approach could evolve over successive generations to specify arbitrary nanotechnology systems including hardware and software.

The first step was to develop a 3rd dimension analogy to Moore’s Law called Optimal Adiabatic Scaling (OAS). We assume manufacturers will become better able to exploit the third dimension for integrated circuits, just as it is being currently exploited for memory. Using adiabatic principles to reduce power consumption, we develop a scaling rule that shows how to build 3D computer systems at constant power consumption per unit area (of a face of a 3D chip).

OAS scaling is not suitable for the von Neumann computer architecture, and we developed a set of principles for where OAS is suitable. Overall computer systems (comprising processor and memory together) and biological systems like brains scale with OAS – even though microprocessors and memories do not scale this way when considered separately.

The central concept is to shift the vision of the computer concept from designs that evolve with human style preferences into a system for efficiently converting physical resources into computing. This would make computers more like electric motors. Electric motors are rated by their power, energy efficiency, reliability, etc. with very little concern about their styling.

PIMS proposes a new type of software that is a 3D arrangement of tiles or blocks, that can be used as either software or a “schematic” for hardware. A PIMS computer would be able to execute the software on the fly, loading the schematic and executing it just as a laptop can load and execute a word processing program. Alternatively, the software could be viewed as a schematic and become a design for implementation in some present or future nanotechnology. Manufacturing a system from a PIMS program would take weeks and result in a computer able to solve only one problem. However, the resulting computer could solve the designated problem very close to the physical limits of performance set by nature.

It is entirely possible that a PIMS computer could run “legacy” software. The PIMS approach is similar enough to conventional computers that existing code could be compiled and executed, with or without “compatibility libraries” to assure legacy software compatibility, albeit without large performance gains. This approach has some analogy to soft-core processors for FPGAs.

PIMS has some advantages at the computer systems level that were explained. Integrating storage and processing can improve I/O bandwidth or sometimes eliminate it altogether (if the data is stored exactly where it is processed). In addition, data can be distributed in analogy to disk striping but with the benefit that it can be processed with greater energy efficiency in the underlying logic. (Note that this is a bit of a mind-stretching concept. It implies, for example, that AND gates and adders become more energy efficient if you store your data on the storage media in a more intelligent way.)

There was a section on instruction set design for PIMS. As mentioned above, the PIMS tiles are the equivalent of instructions. The document gave an example of how to derive the necessary repertoire of tile behaviors based upon the needs of a problem.

We performed an energy efficiency analysis of the PIMS approach in comparison with a current production GPU. With projections based on production DRAM, PIMS outperformed the GPU by about 2000×. If a really far-out adiabatic memory concept could be moved to production, the boost rises to about 80,000×. At 80,000×, the historical rate of improvement due to Moore's Law could be maintained for another 16 generations or so. The energy efficiency analysis illustrated why a hugely stripped down GPU architecture using adiabatic low-level technology would make sense.

The document was in part inspired by an emerging leadership position that a new concept in computing will be needed based on a new device and which offers an exponential scaling path. This document presented a computing concept that will scale to the ultimate limits of nanotechnology, but the author does not consider a new device to be in as much of a central role as indicated. To be specific, the leadership position seems to anticipate that emergence of a new device will be a gating factor for a new concept in computing. The author disagrees, and cites this document as evidence. This document shows that a continued reduction in per-device manufacturing costs will be needed, but that current (CMOS) devices (if integrated with memory) would be sufficient. It seems unlikely that CMOS will be suitable for continued lowering of manufacturing costs, so we need a replacement for CMOS. A CMOS replacement that is cheaper to make is not the same concept as present in the leadership document (for the time being, the leadership document is not public, so the reader will have to take our word for this).

12. References

[Amdahl 67] Amdahl, G.M. Validity of the single-processor approach to achieving large scale computing capabilities. In AFIPS Conference Proceedings vol. 30 (Atlantic City, N.J., Apr. 18–20). AFIPS Press, Reston, Va., 1967, pp. 483–485.

[Bennett 73] C. Bennett, “Logical Reversibility of Computation,” IBM Journal of Research and Development, Volume 17 Issue 6, November 1973, Pages 525-532

[DDR3L] Micron datasheet, DDR3L SDRAM MT41K1G4 – 128 Meg x 4 x 8 banks MT41K512M8 – 64 Meg x 8 x 8 banks MT41K256M16 – 32 Meg x 16 x 8 banks

[Dennard 74] R. H. Dennard, et. al., Design of ion-implanted MOSFET's with very small physical dimensions, IEEE J. Solid-State Circuits, Vol. SC-9, pp. 256-268, 1974.

[Dennis 80] Dennis, J. B. (1980). Data flow supercomputers. *Computer*, *13*(11), 48-56.

[Drexler 92] Drexler, K. E. (1992). *Nanosystems: molecular machinery, manufacturing, and computation*. John Wiley & Sons, Inc.

[Ellson 04] Ellson, John, et al. "Graphviz and dynagraph—static and dynamic graph drawing tools." Graph drawing software. Springer Berlin Heidelberg, 2004. 127-148.

[Feynman 60] Feynman, Richard P. "There's plenty of room at the bottom." Engineering and science 23.5 (1960): 22-36.

[Feynman 96] RP Feynman, JG Hey, RW Allen, Feynman lectures on computation, Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1998

[Frank 2014] D. Frank, “Reversible and Adiabatic Classical Computation: An Overview”

[Fujisaki 13] Y. Fujisaki, Review of Emerging New Solid-State Non-Volatile Memories, Jpn. J. Appl. Phys. 52 040001

[Herculano-Houzel 11] Herculano-Houzel, Suzana. "Scaling of brain metabolism with a fixed energy budget per neuron: implications for neuronal activity, plasticity and evolution." PLoS One 6.3 (2011): e17514.

[HMC 14] Hybrid memory cube is an emerging product by Micron. See the Internet.

[HP The Machine] “The Machine” is an announced research program by Hewlett Packard. See the Internet.

[Isaacson 14] Isaacson, W. (2014). *The Innovators: How a Group of Inventors, Hackers, Geniuses, and Geeks Created the Digital Revolution*. Simon and Schuster.

[ITRS YY] http://www.itrs.net

[Karakiewicz 12] R. Karakiewicz, R. Genov, G. Cauwenberghs, 1.1 TMACS/mW Fine-Grained Stochastic Resonant Charge-Recycling Array Processor, IEEE Sensors Journal, Vol. 12, No. 4, April 2012, p. 785.

[Koller 92] Koller, J.G.; Athas, W.C., "Adiabatic Switching, Low Energy Computing, And The Physics Of Storing And Erasing Information," Physics and Computation, 1992. PhysComp '92., Workshop on , vol., no., pp.267,270, 2-4 Oct 1992
doi: 10.1109/PHYCMP.1992.615554

[Kung 79] Kung, H. T., & Leiserson, C. E. (1979). Systolic arrays (for VLSI). Society for Industrial & Applied Mathematics, 256.

[Landauer 61] Landauer, R. (1961). Irreversibility and heat generation in the computing process. *IBM journal of research and development*, *5*(3), 183-191.

[Moore 65] G. Moore, Cramming More Components onto Integrated Circuits, Electronics pp. 114-117, April 19, 1965.

[Mountain 14] David Mountain, personal discussion and document.

[MT29F512G] Micron data sheet, NAND Flash Memory MT29F64G08CBAA[A/B], MT29F128G08C[E/F]AAA, MT29F128G08CFAAB, MT29F256G08C[J/K/M]AAA, MT29F256G08CJAAB, MT29F512G08CUAAA, MT29F64G08CBCAB, MT29F128G08CECAB, MT29F256G08C[K/M]CAB, MT29F512G08CUCAB

[Nikonov 13] D. Nikonov and I. Young. Overview of Beyond-CMOS Devices and A Uniform Methodology for Their Benchmarking, arXiv 1302.0244.

[Nordhaus 07] W. D. Nordhaus, “Two centuries of productivity growth in computing,” Journal of Economic History, vol. 67, no. 1, p. 128, 2007.

[Simmons 05] Simmons, M. Y., Ruess, F. J., Goh, K. E. J., Hallam, T., Schofield, S. R., Oberbeck, L., ... & Reusch, T. C. G. (2005). Scanning probe microscopy for silicon device fabrication. *Molecular Simulation*, *31*(6-7), 505-515.

[Tail YY] Tail recursion is a well-know aspect of programming. For an introduction and references, see Wikipedia.

[Theis 10] T. Theis, In Quest of the “Next Switch”: Prospects for Greatly Reduced Power Dissipation in a Successor to the Silicon Field-Effect Transistor, Proceedings of the IEEE, Volume 98, Issue 12, 2010

[TN-41-01] Micron technical note, TN-41-01: Calculating Memory System Power for DDR3.

[von Neumann 45] J. von Neumann, First draft of a report on the EDVAC, Moore School of Elec Eng, U. of Pennsylvania, Philadelphia, Pa, June 30, 1945 (101 pp). (This draft was written in March-April 1945.)

[Walter 05] Chip Walter, Kryder's Law, Scientific American (August 2005), 293, 32-33

[Wikipedia YY] http://en.wikipedia.org/wiki/List_of_animals_by_number_of_neurons

Distribution

1	MS0899	Technical Library	9536 (electronic copy)

Sandia National Laboratories