Project CrayOn: Back to the future for a more General-Purpose GPU?

Philip Machanick
Department of Computer Science
Rhodes University
Grahamstown, 6140, South Africa
p.machanick@ru.ac.za

***Abstract*—General purpose of use graphics processing units (GPGPU) recapitulates many of the lessons of the early generations of supercomputers. To what extent have we learnt those lessons, rather than repeating the mistakes? To answer that question, I review why the Cray-1 design ushered in a succession of successful supercomputer designs, while more exotic modes of parallelism failed to gain traction. In the light of this review, I propose that new packaging breakthroughs create an opening for revisiting our approach to GPUs and hence GPGPU. If we do not do so now, the *GPU endpoint* – when no GPU enhancement will be perceptible to human senses – will in any case remove the economic incentive to build ever-faster GPUs and hence the economy of scale incentive for GPGPU. Anticipating this change now makes sense when new 3D packaging options are opening up; I propose one such option inspired by packaging of the Cray-1.**

I. Introduction

Supercomputers and in general high-performance computing (HPC) designs evolved from faster versions of conventional designs through vector processors to increasingly exotic modes of parallelism. Eventually, options shook down to two: shared-memory multiprocessors with conventional instruction sets and vector machines. As each new wave of packaging advance made it possible to move functionality to lower-cost designs, some of these design choices were recapitulated.

The Cray-1 represents an interesting inflection point; while not the first vector machine, it was the first to be commercially successful. To some extent, it built on Seymour Cray's prior experience at Control Data and lessons from earlier vector machines but a critical factor in its success is the fact that a relatively large-scale semiconductor memory had just become possible, displacing magnetic core memories made of discrete components, and hence inherently hard to scale to large sizes [1]. Many exotic modes of parallelism were explored in the early supercomputer market; Cray relied on methods that are now mainstream: pipelines, multiple issue and multiple CPUs, with vectors the only significant instruction-level difference from earlier Control Data designs [2].

Given that the Cray-1 was such a success, it is informative to re-examine current assumptions and trends in the light of why Cray was able to make such a big leap ahead. In today's market, the parts most similar to the weird and wonderful designs from the early supercomputer era are graphics processor units (GPUs). If major packaging changes are imminent, it is useful to examine whether charging ahead with more of the same is the best option.

Generally, reasons that more exotic modes of parallelism failed have not changed. If an approach was hard to program in a multi-million-dollar multi-chip design, it remained hard to program when reduced to much less expensive hardware. There is no Moore's Law for programming skills. The only thing that has changed is that in recent years, very aggressive, complex modes of parallelism have found a mass market in the form of GPUs and so general-purpose use of these parts (GPGPU) has taken on a new life because there is a natural mass market for GPUs. However, eventually, what I call the *GPU endpoint* will arise: no improvement in GPUs will have a noticeable effect on human perception. At that point, a faster GPU will have no mass market and hence no clear justification for use in HPC other than momentum.

I examine here whether there are design alternatives for GPUs that could make for a more general-purpose GPU, i.e., one that was easier to program for a variety of workloads. The benefit of such a design is that a part targeting high-end computation without a natural mass market is difficult to bring to scale. This idea is not entirely novel: Intel partially explored this design space with the discontinued Larrabee project [3]. However, Intel started from a base of a Pentium pipeline; I argue here that a real back-to-basics look at what was successful in the early supercomputer market will provide a better starting point for a more general-purpose GPU. Designs of the early-RISC era delivered more performance for like chip area than CISC competitors. The RISC advantage decreases with more aggressive designs: the complex front-end needed for a CISC architecture becomes a proportionately smaller overhead. This paper is part of a project called CrayOn[1] that reexamines the GPGPU concept starting from a platform more amenable to general-purpose programming [4].

I review the history of supercomputers, then map this history onto shifts in packaging that enable new niches to emerge. From this, I propose an alternative view of GPU design with an

The financial assistance of the South African National Research Foundation (NRF) towards this research is hereby acknowledged. Opinions expressed and conclusions arrived at, are those of the authors and are not necessarily to be attributed to the NRF. NRF/CPRR Grant 98952.

[1] The CrayOn name is inspired by "Cray On a chip" but, for trademark reasons, officially refers to another kind of graphics device, a crayon.

eye on programming models that have proved to be relatively adaptable to general-purpose computation.

II. Origins of Supercomputers

In general, performing above the norm requires some way of processing faster than a conventional design. These variations include more expensive memory, more instruction-level parallelism (ILP) and various other modes of parallelism that deviate from a conventional instruction set including variants on single instruction multiple data stream (SIMD) of which vectors are a specific case.

A. From Control Data to Cray-1

The Cray-1 was the first successful vector supercomputer [5]. Its biggest single win over earlier vector designs was relatively short start latencies for vector operations meaning that the crossover point for vector operations to be faster than scalar were as short as 2–4 elements [5]. A particular feature of the Cray-1 design was the use of static RAM [1]; in order to achieve reasonable density and minimal propagation delays, Cray adopted their signature compact cylindrical layout [5].

B. Expansion to Weird and Wonderful

In the 1980s, there was extensive work on distributed-memory multiprocessors, with a wide range of interconnection strategies [6]. Much of this work was characterised as massively parallel processing (MPP) – the idea was that exploiting parallelism of the order of hundreds or even thousands removed the need for exotic CPU technology. The most extreme example of this type was the Connection Machine (CM). In its original form, the CM-1, arithmetic used bit-serial ALU operations and it relied on massive parallelism (16Ki–64Ki nodes) to compensate for relatively slow operations [7].

The INMOS transputer (spelt without an initial capital, in the supposition that it would become a common noun) was specifically designed for this style of design with dedicated high-speed serial links and even a similarly lowercased programming language, occam, specialized to this purpose [8].

One aspect of distributed-memory designs particularly with relatively low-powered CPUs was the needed for a scalable interconnect. CM-5 used a *fat tree*, with higher channel capacity the higher a switch is in the tree. CM-5 could have as few as 32 Sparc CPUs and a 64-wide vector unit (the initial idea of numerous slow CPUs by then abandoned) and as many as 65,536. [9]. Hypercube-based interconnects were a popular choice [10]. A reason for that is that other networks such as trees of butterfly networks map to hypercubes [11]. One hypercube machine of that era is nCUBE [12].

Orthogonal to the question of where memory sits is the question of control – is a multiprocessor system MIMD, SIMD or some other variant? CM attempted to straddle a boundary by running the same code in parallel across all nodes with a separate control network to coordinate the CPUs: it was using conventional CPUs that each ran the same code with the effect of SIMD [9]. Another SIMD design is the ICL DAP [13] and a popular textbook of the era refers extensively to SIMD and numerous designs that have not survived [14].

C. The VLIW excursion

Multiflow took supercomputers a different direction with Very Long Instruction Word (VLIW). The idea: group multiple instructions into a single long word to increase ILP. In Multiflow's design, up to 28 simultaneous operations could be encoded in a single wide instruction word that could control every functional unit [15], [16], inspired by the concept of horizontal microcode [17].

VLIW relies on a compiler to find fine-grained parallelism [15]. That is conceptually similar to a superscalar architecture without dynamic dispatch, with the drawback of wasting code space if some instruction words cannot be fully utilized [18]. Given that studies on limits of instruction-level parallelism for static instruction scheduling under realistic assumptions, around the time when Multiflow ceased operations in March 1990 [19], showed fine-grained parallelism to be less than 10 in most workloads [18], [20], it is questionable whether the level of parallelism Multiflow aimed for was realistic.

D. Darwinian Selection

The critical question now is: which of these have survived? And which of these could work as a the starting point of a new wave of designs based on emerging design trade-offs? Large-scale distributed-memory machines with exotic interconnects are a thing of the past. "Transputer" is not a common noun. VLIW still exists in the form of the Itanium range (IA-64), though it has failed to establish a big market. Vector instruction sets still exist in various forms though no one makes a vector machine comparable to the original Cray line.

There are two explanations for the demise of the various more specialist designs. The first is that conventional CPUs have become increasingly competitive and the second is that the more exotic approaches tried in the past were too hard to program. There is a plethora of literature about programming these machines often with a single paper devoted to a single algorithm or even a single kernel.

III. The next wave: single-chip designs

A. VLIW revisited

Bob Colwell, of the Multiflow design team, was at Intel when Intel embarked on the IA-64 VLIW project. Colwell, instead of working on IA-64, worked on the next-generation Pentium, which was to lead to a series of successful designs that in effect rescued Intel when IA-64 failed to deliver at the level required to replace IA-32 as a mass-market part. Did Colwell know something that Intel strategists didn't?

IA-64 has 128 general-purpose and 128 floating-point registers and many instructions specify a predicate register. Three 41-bit instructions are packed into a 128-bit word; the remaining 5 bits used to guide the hardware on available parallelism. IA-64 also introduced speculative loads [21]. The philosophy follows on from Multiflow, if with a lower expectation of statically-identifiable parallelism: the compiler is expected to do more of the work than in other similar-era designs in finding ILP, reducing hardware complexity at least at the level of instruction scheduling. IA-64 was claimed to exemplify "explicitly parallel instruction computing", or EPIC [22].

For IA-64 to have succeeded, it needed to be competitive where Multiflow wasn't with other approaches to higher ILP, particularly dynamic dispatch in multiple-issue pipelines. While its 3-instruction bundle parallelism was relatively modest, the requirement that the compiler identify the parallelism flew in the face of evidence that it is difficult to statically schedule particularly in the presence of branches. While predicated instructions can help with small basic blocks, it was brave of Intel to venture in where Multiflow had failed – particularly as they had inherited Colwell who can reasonably be supposed to have learnt a thing or two at Multiflow.

B. HLL Architectures Redux: Or Not?

High-Level Language (HLL) architectures provide some insights into how ideas do not necessarily transfer across technology eras. One of the first computers I used was a Burroughs B5700. The Burroughs B5000 series was the first of a range of successful HLL models. They featured a number of innovations that worked well in their day. They used variable-sized segments rather than pages as their unit of virtual memory allocation, they used stack-based arithmetic and had hardware support for arrays, which were implemented as segments, with descriptors that made hardware bounds checking possible. There was also hardware support for basic data types with memory tags indicating types so a single instruction could work for multiple data types [23].

Two factors made it possible for all the design choices to be effective. The B5000 appeared when core memories made it possible to implement sophisticated operating systems and programming languages without running out of memory. At the same time, memories were not so large that it was a reasonable trade-off to waste memory to gain speed. Secondly, The B5000 design was a masterclass in hardware-software co-design, with compiler, hardware and operating system designed in close cooperation [24]. In a remarkable innovation for its time, the machine did not even have an assembler; the operating system was written in Balgol, a variant on Algol 60 designed to support systems programming [24].

Subsequent attempts at HLL architectures were only successful in niches (e.g., LISP machines had a following [25]).

Exemplifying this failure is Intel's iAPX 432. This was an ambitious project to design a microprocessor with extensive HLL support. Performance was generally poor [26], with Intel's own much more mundane 80286 an order of magnitude faster [27]. The VAX, though less strongly HLL-oriented, had a complex ISA that aimed to support compiler writers. While it sold in reasonable volumes, its performance was uncompetitive with comparable RISC designs [28]. By the late VAX era, the first advantage of HLL architectures – code compactness – ceased to be a major factor. In one example, the complex VAX `CALL` instruction can be replaced by a sequence of simpler instructions with an runtime saving of 20% [29]. The second Burroughs advantage, hardware-software co-design, is useful with any variant on ISA design and is not enough to rescue HLL architectures now that memory footprint is not the major factor it was in earlier eras.

IV. Back to the Future

The point of this historical excursion is to search for nuggets to rework for the current context. What major lessons can we learn from the past? What are current constraints, and how do they fit historical lessons? Where does that take us?

A. Key Lessons

An idea that is inherently bad is not improved by better packaging. An idea that only worked because of packaging constraints of a different era will not work now, unless design trade-offs recapitulate the design constraints of that era.

I reduce "inherently bad" to simple questions addressing usability of ISA features and broad architectural design choices:

- Can we program it? That further breaks down to:
  - Is is a fit to current programming languages?
  - Does it fit a reasonable range of reasonably accessible algorithms?
  - Does it avoid sticking points of past failed designs like programmer-controlled specialist memory?
  - Can a compiler generate good code for it without major programmer intervention?
  - Does it fit current operating system designs?
- Does it generalize beyond today's design constraints?
  - Does its advantage dissipate if the ground shifts?
  - If we project sustainable trends over ten years does it still work?
  - Does it require coding changes as hardware improves?

Can we program it? New programming languages are always appearing but the bar to acceptance is far higher if a new language is a prerequisite to using a new architecture. Cray-1 was a good fit to existing languages with modest programming adjustments to achieve good results from vectorising. There is no example I know of where an architecture requiring a fundamental shift in programming language has succeeded.

The Connection Machine and other exotic modes of parallelism failed despite using conventional languages because too few algorithms were a good fit to their programming model. In general, architectures with a high ratio of communication to computation failed on scaling up to large numbers of processors. Programmer-controlled specialist memory is a similar issue. Overheads of copying have to be amortized by a high enough amount of computation. While tricks like DMA and double-buffering can hide the overheads of copying, they impose a significant extra programming load and also may require redesign if the relative sizes or speeds of memories change, an issue we thought we solved with virtual memory. In one example, a matrix multiply on the Cell Broadband Engine with a 256kiB local memory for its vector units has over 8000 lines of assembly language [30]. In general this small local store presented significant programming challenges, leading to a flurry of papers on how best to program the device of which I cite a small sample [31]–[33].

While programming experience with the Cell architecture developed over time, in general this type of architecture has turned out to be difficult to program without specialist algorithms despite the emergence of toolkits like OpenCL [34]. If compilers could generate good code for designs like the Cell (and for that matter, GPUs), there would be no need for specialist toolkits and libraries like OpenCL and CUDA [35].

What of operating system support? The programming model of the Cell and GPUs is a poor fit to the virtual memory view of a single uniform address space. Here are a few examples of the need for an architectural innovation to fit OS constraints cleanly:

- my RAMpage idea of moving VM up a layer to replace the lowest-level cache with the main memory, making DRAM a paging device backed by a slower layer of backing store [36]
- the core fusion idea of designing a CPU that could be dynamically reconfigured as either a single aggressive pipeline or multiple less aggressive cores [37]
- simultaneous multithreading (SMT) [38], implemented by Intel as Hyperthreading [39]

RAMpage looked good in simulations but implementing it would have required an OS aware of the new memory hierarchy and that did not happen. Core fusion would have required an OS to be able to adjust dynamically to a different number of CPUs with different capabilities and that too did not eventuate. Finally, SMT is a counter-example, with significant operating system work on benefiting from the new feature [40]. In the case of SMT, the variation required was not too onerous – since modern kernels general have kernel-level thread support, adjusting for a different form of threads was not a major change.

Does it generalize beyond today's design constraints? The HLL movement exemplifies over-specialization to constraints of a given era: the early success of Burroughs was not replicated by later designs. While some of that may be explained by lesser hardware-software co-design, memory constraints in which the Burroughs B5000 and successors operated no longer applied to later designs. The RISC movement took off largely because bigger memories made small memory footprint a second-class concern outside niches like embedded systems.

Programmer-controlled memory hierarchy is another example that ages poorly. As memories become cheaper, the algorithmic and data structure constraints change and investments in coding to a specific memory size are lost.

B. Current Constraints

A growing constraint on packaging is how much can be fitted into a single die. Moore's Law does not prevent wider problems of scaling even when it is possible to continue increasing the component count at the historical rate. There are issues such as leakage current (a concern for more than 10 years [41]) and limits to ILP that have led to moves like chip multiprocessors [42] (or multicore in commercial parlance). Another factor is the memory wall [43], an issue to some extent mitigated by large on-chip caches and the reduced rate of clock speed increase in recent years.

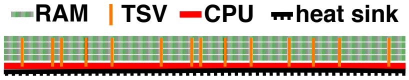

Fig. 1. 3D die stacking with CPU and RAM in one package.

These pressures add up to a more 3-dimensional view of chip design.

One approach to going 3-dimensional is 3D die stacking, which has the advantage that dissimilar technologies can relatively easily be combined. For example, PicoServer was a design study that combined a multicore CPU layer with DRAM layers, using through-silicon vias (TSVs) to create a package that eliminated off-chip latencies. Figure 1 illustrates the general idea. The advantage of this form of die stacking is that it can utilize standard technologies to build a very compact system and the reduced latencies mean that the system can run at lower clock speed for a given level of performance, hence saving energy [44]. The drawback of this approach is that heat dissipation limits the number of layers and potential overall speed. Possibly for this reason, more recent work has focused on building fast RAM by including a logic layer in the 3D stack as in Hybrid Memory Cube (HMC), which stacks DRAM on top of a logic layer designed to give faster access to DRAM rather than including a CPU layer in the stack [45].

C. Where Next?

Stacking as a model of going 3D assumes a planar universe in which another dimension can only be introduced by adding another layer; this is more like a 2.5-dimensional universe than a 3D universe. While there are some genuinely 3D technologies like 3D Xpoint nonvolatile RAM (NVRAM), where the 3D structure plays a role in the memory design [46] and there are moves to implement logic structures in 3 dimensions [47], the simplest way of going 3D is to combine dies. Is 2.5D really the best option?

Another alternative is to build a 3D structure out of dies that utilizes all three dimensions more fully. I take inspiration from Cray's idea of organizing components around a cylinder to minimize propagation delay. Consider Figure 1 and now reconceptualise it as interconnected on a semicircular bus instead of TSVs. The result is illustrated in Figure 2. Once you have one module like this, a simple generalization is to allow combination of such packages to build a larger system. This packaging has several advantages:

- *heat dissipation* – the components are no longer packed close together except near the interconnect and there is space on the inside of the arc for heat dissipation
- *reusable components* – provide off-chip connections can be brought to one edge of the die for this scenario, the individual dies can be reused in other packaging
- *scalable*
  - the number of components in one package can be increased to the limit of removing heat

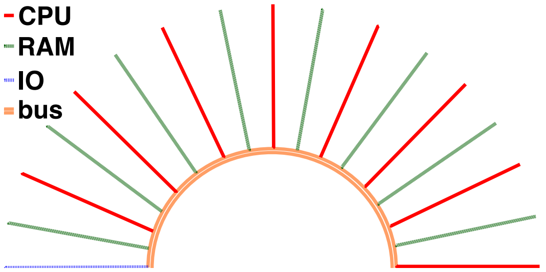

Fig. 2. Cray-1 Redux: 3D packing on an arc.

  - neighbouring packages, with higher latency to access, can be used to build a larger system
- *generalizable* – there is no reason to have a fixed ratio of RAM to CPU; even the number of IO layers could vary

The major requirement for this idea to work is manufacturing the interconnect to sufficient accuracy to remove the need for inter-chip buffering. That should not be more complex than die stacking, where TSVs need to be placed with high accuracy – and each die needs to be designed so that its nearest neighbour has conductors in the required spot.

Once we have this idea, there is no reason that the individual elements need be single-layer dies, provided the layering does not compromise heat dissipation. For example, the RAM components could be HMC for high density and low latency.

V. Putting it all together

New 3D options allow us to combine memory and computation with a low-latency interconnect, closer to the Cray-1 universe than current packaging. What does that buy us? Can we do a better (more general-purpose) GPU and if so, why is the timing for that right now?

A. A better GPU

Returning to my starting point, would it make sense to design a GPU that looked architecturally more like a Cray 1, with direct support for parallelism only in the form of ILP and vectors? Any other parallelism mode added would have to pass a test of being easy to use in general-purpose code. The result: a much more appealing programming model for general-purpose computation. That Intel's Larrabee design was uncompetitive with conventional GPUs does not mean it is impossible to do better.

The changes versus Larrabee that I propose are:

- *a simple RISC ISA* – e.g., RISC-V is unencumbered and is based on lessons from earlier RISC designs [48]
- *minimize arcane modes of parallelism* – if you can't program it easily, find an easier alternative
- *exploit advances in memory technology* – HMC and faster NVRAMs for storage make for fast overall design
- *innovative 3D interconnection* – my Cray-1-style layout or any other that minimizes latencies while allowing heat dissipation makes it easier to achieve a balanced design
- *remember other software layers* – an easy fit to existing operating systems is the optimum choice in today's world. Despite the benefits of designing a complete system from scratch using hardware-software co-design, the practical reality is that it is far easier to build on existing compilers, tools and operating systems

B. The GPU endpoint

GPUs have a large captive market in which the sole requirement for acceptance once hardware interfacing is taken care of is designing new drivers for all platforms of interest. That gives GPU designers latitude to break the rules. Unfortunately, that also means that GPUs need not be designed to be as easy to program as general-purpose parts. My central thesis is that this need not be the case – the approach I outline here could lead to a GPU that is in fact good for more general algorithms, rather than a part that justifies considerable pain to use because it's there and relatively inexpensive.

Beyond the GPU endpoint, a more powerful GPU is a waste and engineering ever-more powerful GPUs purely as computation engines will no longer have the support of a mass-market application. Such parts *will not gain from economy of scale* while still being *hard to program.*

C. Decision Point

We can wait for the GPU endpoint. After that, modes of parallelism less optimal for graphics will become more defensible in a GPU. If a GPU can render way faster than human perception, who cares if it runs 10% slower as the price for a more general programming model? Or we can start now, and reconceptualize the GPU so it is a more tractable general-purpose compute engine.

I prefer the latter choice as it is more sustainable; we already know from previous generations of supercomputer what computational modes are applicable to the widest range of problems. Why hurt ourselves by constraining the solution space to modes that work best for graphics pipelines?

You could argue that this is all an unnecessary as there is nothing to constrain architects from designing more general HPC architectures. Intel has taken that view in abandoning the Larrabee experiment in favour of the Xeon Phi which, in a recent version, has 72 cores, each with two vector units. It also supports two types of DRAM, much like a GPU in which there is specialist high-speed RAM, and it has memory features to support vector instructions [49]. Xeon Phi is a step closer to my vision for an alternative GPU [4] except that it still uses the old Intel ISA as its base, which limits performance scalability of non-vector computation. Intel has abandoned the idea of marketing such a part as a GPU, so that also eliminates the advantage of a mass market to build economy of scale.

GPGPU is not a bad idea. As long as there is a mass market for GPUs, why not use them for other purposes? A mass market for a part that is also useful for HPC has very appealing economics versus a part that is only good for a low-volume market like HPC. That being the case, I argue for designing a GPU starting from a more general-purpose design.

Why now? Because GPUs will continue to evolve for long enough to matter and because the packaging trends I outline in this paper create the opportunity re-evaluate design trade-offs.

References

[1] E. Normand, J. L. Wert, H. Quinn, T. D. Fairbanks, S. Michalak, G. Grider, P. Iwanchuk, J. Morrison, S. Wender, and S. Johnson, "First record of single-event upset on ground, Cray-1 computer at Los Alamos in 1976," *IEEE Transactions on Nuclear Science*, vol. 57, no. 6, pp. 3114–3120, 2010.

[2] R. L. Sites, "An analysis of the Cray-1 computer," in *Proc. 5th Annual Symposium on Computer Architecture*, 1978, pp. 101–106.

[3] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin *et al.*, "Larrabee: a many-core x86 architecture for visual computing," in *Proc. SIGGRAPH '08*, 2008.

[4] P. Machanick, "How general-purpose can a GPU be?" *South African Computer Journal*, no. 57, pp. 113–117, 2015.

[5] R. M. Russell, "The CRAY-1 computer system," *Communications of the ACM*, vol. 21, no. 1, pp. 63–72, 1978.

[6] R. A. Finkel and M. H. Solomon, "Processor interconnection strategies," *IEEE Transactions on Computers*, no. 5, pp. 360–371, 1980.

[7] L. W. Tucker and G. G. Robertson, "Architecture and applications of the Connection Machine," *Computer*, vol. 21, no. 8, pp. 26–38, August 1988.

[8] C. Whitby-Strevens, "The transputer," in *Proc. 12th Annual International Symposium on Computer Architecture*, 1985, pp. 292–300.

[9] C. E. Leiserson, Z. S. Abuhamdeh, D. C. Douglas, C. R. Feynman, M. N. Ganmukhi, J. V. Hill, D. Hillis, B. C. Kuszmaul, M. A. St Pierre, D. S. Wells *et al.*, "The network architecture of the connection machine CM-5," in *Proc. fourth annual ACM symposium on Parallel algorithms and architectures*, 1992, pp. 272–285.

[10] J. P. Hayes, T. Mudge, Q. F. Stout, S. Colley, and J. Palmer, "A microprocessor-based hypercube supercomputer," *IEEE Micro*, vol. 6, no. 5, pp. 6–17, 1986.

[11] O. A. McBryan, "An overview of message passing environments," *Parallel Computing*, vol. 20, no. 4, pp. 417–444, 1994.

[12] M. M. Waldrop, "Hypercube breaks a programming barrier," *Science*, vol. 240, no. 4850, pp. 286–287, 1988.

[13] D. Parkinson and J. Litt, *Massively parallel computing with the DAP*. MIT Press, 1990.

[14] R. W. Hockney and C. R. Jesshope, *Parallel Computers 2: architecture, programming and algorithms*. CRC Press, 1988.

[15] R. P. Colwell, R. P. Nix, J. J. O'Donnell, D. B. Papworth, and P. K. Rodman, "A VLIW architecture for a trace scheduling compiler," *IEEE Transactions on computers*, vol. 37, no. 8, pp. 967–979, 1988.

[16] R. P. Colwell, W. E. Hall, C. S. Joshi, D. B. Papworth, P. K. Rodman, and J. E. Tornes, "Architecture and implementation of a VLIW supercomputer," in *Proc. 1990 ACM/IEEE conference on Supercomputing*. IEEE Computer Society Press, 1990, pp. 910–919.

[17] S. Stritter and N. Tredennick, "Microprogrammed implementation of a single chip microprocessor," in *Proc. 11th Annual Workshop on Microprogramming*, ser. MICRO 11. Piscataway, NJ, USA: IEEE Press, 1978, pp. 8–16. [Online]. Available: http://dl.acm.org/citation.cfm?id=800132.804299

[18] N. P. Jouppi and D. W. Wall, "Available instruction-level parallelism for superscalar and superpipelined machines," in *Proc. Third International Conference on Architectural Support for Programming Languages and Operating Systems*, 1989, pp. 272–282.

[19] E. Fisher, *Multiflow Computer: A Start-up Odyssey*. CreateSpace Independent Publishing Platform, 2013.

[20] D. W. Wall, "Limits of instruction-level parallelism," in *Proc. Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 1991, pp. 176–188.

[21] J. Huck, D. Morris, J. Ross, A. Knies, H. Mulder, and R. Zahir, "Introducing the IA-64 architecture," *IEEE Micro*, vol. 20, no. 5, pp. 12–23, 2000.

[22] L. Gwennap, "Intel, HP make EPIC disclosure," *Microprocessor report*, vol. 11, no. 14, pp. 1–9, 1997.

[23] A. J. W. Mayer, "The architecture of the Burroughs B5000: 20 years later and still ahead of the times?" *SIGARCH Comput. Archit. News*, vol. 10, no. 4, pp. 3–10, Jun. 1982.

[24] W. Lonergan and P. King, "Design of the B5000 system," *Datamation*, vol. 7, no. 5, pp. 28–32, 1961.

[25] D. A. Moon, "Architecture of the Symbolics 3600," in *Proc. 12th Annual International Symposium on Computer Architecture*, 1985, pp. 76–83.

[26] P. M. Hansen, M. A. Linton, R. N. Mayo, M. Murphy, and D. A. Patterson, "A performance evaluation of the Intel iAPX 432," *ACM SIGARCH Computer Architecture News*, vol. 10, no. 4, pp. 17–26, 1982.

[27] D. A. Patterson, "A performance evaluation of the Intel 80286," *ACM SIGARCH Computer Architecture News*, vol. 10, no. 5, pp. 16–18, 1982.

[28] D. Bhandarkar and D. W. Clark, "Performance from architecture: Comparing a RISC and a CISC with similar hardware organization," in *Proc. Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 1991, pp. 310–319.

[29] D. A. Patterson, "Reduced instruction set computers," *Communications of the ACM*, vol. 28, no. 1, pp. 8–21, 1985.

[30] D. Hackenberg, "Fast matrix multiplication on Cell (SMP) systems," 2007, last accessed: 28 December 2017. [Online]. Available: http://www.tu-dresden.de/zih/cell/matmul

[31] J. Kurzak, W. Alvaro, and J. Dongarra, "Optimizing matrix multiplication for a short-vector SIMD architecture–CELL processor," *Parallel Computing*, vol. 35, no. 3, pp. 138–150, 2009.

[32] J. Kurzak and J. Dongarra, "QR factorization for the Cell Broadband Engine," *Scientific Programming*, vol. 17, no. 1-2, pp. 31–42, 2009.

[33] M. Gschwind, "The cell broadband engine: exploiting multiple levels of parallelism in a chip multiprocessor," *International Journal of Parallel Programming*, vol. 35, no. 3, pp. 233–262, 2007.

[34] S. Rul, H. Vandierendonck, J. D'Haene, and K. De Bosschere, "An experimental study on performance portability of OpenCL kernels," in *2010 Symposium on Application Accelerators in High Performance Computing (SAAHPC'10)*, 2010. [Online]. Available: https://biblio.ugent.be/publication/1016024

[35] NVIDIA, "NVIDIA Cuda C programming guide," 2015, last accessed: 28 December 2017. [Online]. Available: http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf

[36] P. Machanick, P. Salverda, and L. Pompe, "Hardware-software trade-offs in a Direct Rambus implementation of the RAMpage memory hierarchy," in *Proc. 8th Int. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS VIII)*, 1998, pp. 105–114.

[37] E. Ipek, M. Kirman, N. Kirman, and J. F. Martinez, "Core Fusion: Accommodating software diversity in chip multiprocessors," in *Proc. 34th Annual International Symposium on Computer Architecture*, 2007, pp. 186–197.

[38] D. M. Tullsen, S. J. Eggers, and H. M. Levy, "Simultaneous multithreading: Maximizing on-chip parallelism," in *Proc. 22nd Annual International Symposium on Computer Architecture*, 1995, pp. 392–403.

[39] D. Koufaty and D. T. Marr, "Hyperthreading technology in the Netburst microarchitecture," *IEEE Micro*, vol. 23, no. 2, pp. 56–65, 2003.

[40] J. R. Bulpin and I. A. Pratt, "Multiprogramming performance of the Pentium 4 with Hyper-Threading," in *Second Annual Workshop on Duplicating, Deconstruction and Debunking*, 2004, pp. 53–62.

[41] N. S. Kim, T. Austin, D. Baauw, T. Mudge, K. Flautner, J. S. Hu, M. J. Irwin, M. Kandemir, and V. Narayanan, "Leakage current: Moore's law meets static power," *computer*, vol. 36, no. 12, pp. 68–75, 2003.

[42] K. Olukotun, B. A. Nayfeh, L. Hammond, K. Wilson, and K. Chang, "The case for a single-chip multiprocessor," in *Proc. Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, 1996, pp. 2–11.

[43] W. A. Wulf and S. A. McKee, "Hitting the memory wall: implications of the obvious," *ACM SIGARCH Computer Architecture News*, vol. 23, no. 1, pp. 20–24, 1995.

[44] T. Kgil, A. Saidi, N. Binkert, S. Reinhardt, K. Flautner, and T. Mudge, "PicoServer: Using 3D stacking technology to build energy efficient servers," *J. Emerg. Technol. Comput. Syst.*, vol. 4, no. 4, pp. 16:1–16:34, Nov. 2008.

[45] R. Courtland, "Memory in the third dimension," *IEEE Spectrum*, vol. 51, no. 1, pp. 60–61, January 2014.

[46] K. Bourzac, "Has Intel created a universal memory technology?[news]," *IEEE Spectrum*, vol. 54, no. 5, pp. 9–10, 2017.

[47] J. Cartwright, "Intel enters the third dimension," *Nature News*, 2011.

[48] K. Asanović and D. A. Patterson, "Instruction sets should be free: The case for RISC-V," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2014-146, 2014, last accessed: 28 December 2017. [Online]. Available: https://www2.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-146.pdf

[49] A. Sodani, R. Gramunt, J. Corbal, H.-S. Kim, K. Vinod, S. Chinthamani, S. Hutsell, R. Agarwal, and Y.-C. Liu, "Knights landing: Second-generation Intel Xeon Phi product," *IEEE Micro*, vol. 36, no. 2, pp. 34–46, 2016.