Array Languages and the Challenge of Modern Computer Architecture

Paul Cockshott

There has always been a close relationship between programming language design and computer design. Electronic computers and programming languages are both ’computers’ in Turing’s sense. They are systems which allow the performance of bounded universal computation. Each allows any computable function to be evaluated, up to some memory limit. This equivalence has been understood since the 30s’ when Turing machines (Turing 1937) were shown to be of the same computational power as the λ calculus.

The initial relationship between languages and machines was one of logical equivalence, but practical distinction. The designers of the first generation computers like the Manchester Mark 1 (Lavington 1980), or Turing’s ACE were concerned with the practical task of getting a machine to compute for the first time.

Stack machines

During the 1950’s the first programming languages COBOL, FORTRAN and ALGOL came to be developed. The existence of these languages affected the design of the second and third generation computers. One minor effect was to ensure that decimal arithmetic was supported in machines like the 360(Emerson W. Pugh and Palmer 1991) in order to accommodate the needs of COBOL compilers. A more significant influence was the invention of stack machines by Burroughs (Lonergan and King 1985) in 1960. These were designed to support block structured languages like ALGOL 60 (Naur and Backus 1960). The influence of the B5000 and ALGOL persisted into the 1970s when the Intel 8086 (Morse, Ravenel, Mazor and Pohlman 1985) was released.

The 8086 incorporated a stack based procedure calling mechanism designed to optimise the implementation of block structured languages of the Algol family, though given the date at which it was released, Pascal rather than Algol would have been the intended language. Shortly afterwards Intel released the stack oriented floating point processor 8087(Palmer and Morse 1984) that supported a reverse polish notation for floating point arithmetic. The availability of the 8087 chip on the original IBM PC was a critical factor in enabling IBM to launch an efficient APL interpreter on that machine. The Pentium processors in modern PCs retain the basic stack oriented arithmetic unit of the 8086 and the Pascal call mechanisms of the 8086.

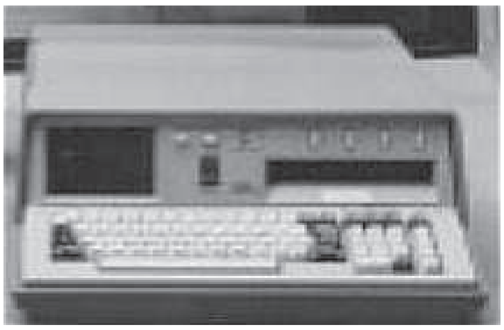

Figure 1: The IBM 5100 micro-coded APL microcomputer of 1975.

APL machines

During the 1970s attempts were made to directly implement high level languages in hardware most famously with the SYMBOL (Smith, Rice, Cheskey, Laliotis and Lundstrum 1985) computer which incorporated an interpreter and virtual storage manager for a heap based programming language directly in the hardware. In addition micro-coded APL machines were proposed or implemented (Abrams 1970, Hassitt, Lageshulte and Lyon 1973, Hakami 1975). The most widely used microcoded APL machine was the IBM 5100 which was a precursor of the IBM PC. The aim of these machines was to narrow the semantic gap between machine code and the language used by applications programmers. Improved performance was a consideration but not the primary one. The 5100 had only an 8 bit ALU and had a performance comparable to other 8 bit micros.

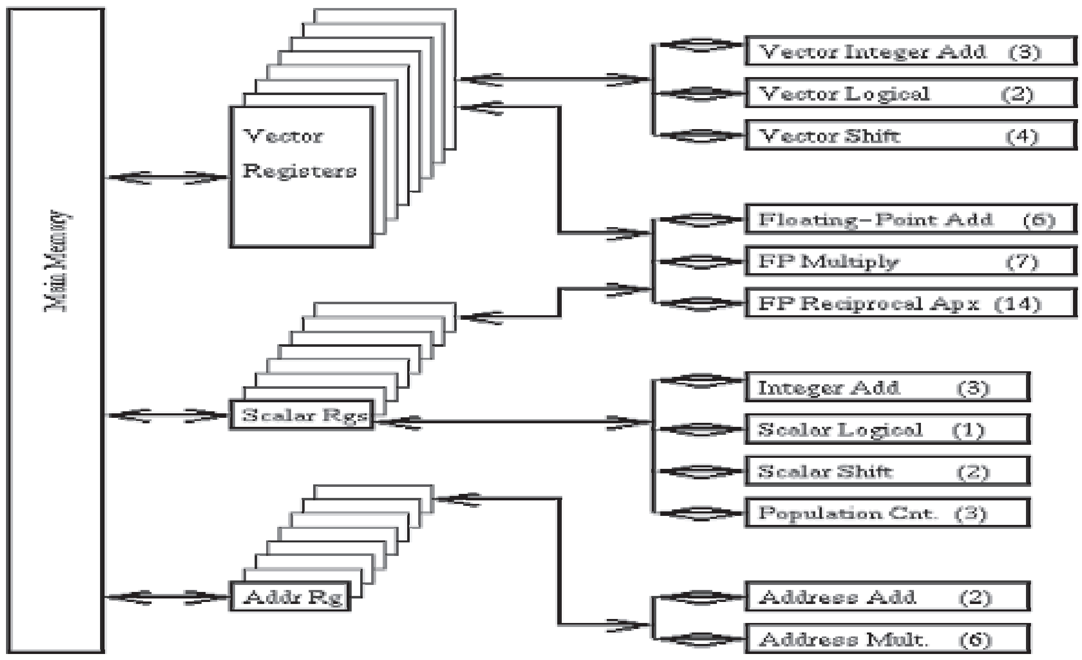

Figure 2: Basic Cray 1 architecture showing vector registers.

Super computers

Another strand of development at this stage was the invention of vector and SIMD super-computers. These aimed, above all, at getting the highest possible performance. Each represented a different approach to overcoming the bottleneck imposed on conventional computers by their need to fetch and update variables in memory a word at a time.

The vector processors pioneered by Cray (Russell 1985) based themselves on the principle of pipelining vector operations with the aim of delivering one floating point result every clock cycle. Instructions operated between vector registers, each of which could hold 64 floating point values. Thus a vector instruction like

V1=V2+V3

would add the corresponding elements of vectors V2 and V3 storing the result in the register V3. The use of pipelining enabled the results to be delivered one every 10ns, which was astonishingly fast at that time. With care it was possible to break up complex FORTRAN expressions in such a way that the intermediate results of an expression would be stored in vector registers. Only the initial loading and storing of the V registers required access to main memory, and this was done by what amounted to fast DMA transfers. It was found that if memory accesses took the form of streaming transfers, the effective bandwidth of the memory could be much faster than for random access transfers. A streaming transfer reduced the set-up time needed to transmit addresses and also allowed effective use of bank interleaving to reduce memory latency.

Compare to this the approach adopted by ICL, whose Distributed Array Processor (Reddaway 1973) or DAP used 4096 small processors operating in parallel. Each processor had a simple 1 bit arithmetic unit to which a single 1 bit wide RAM chip was proximally connected. These processing units were so arranged that on each clock cycle each and every one of them performed the same logical or arithmetical operation. If you were to hold in your mind an image of a battalion of the Guards performing square drill, each foot-soldier moving in perfect synchrony to the regimental drum under the orders of the Sergeant Major, you would then have a perfect metaphor for the operation of the DAP. The DAP operated as an auxiliary processor to a conventional 2980 mainframe with the DAP memory array forming part of the general purpose memory of the 2980. The fact that each processor could operate on but one bit, was circumvented by their performance of bit serial arithmetic. Alternatively the processors could be grouped in blocks of 8, 16, 32 or 64 to perform wider arithmetic. Thus in the widest mode the DAP could operate on 64 x 64bit numbers each step.

The ability to tailor the word length to the application was particularly useful in graphics where one operated on arrays of bits or bytes. The DAP architecture and that of Hillis's (Hillis 1986) derivative Connection Machine was referred to as SIMD - Single Instruction Multiple Datastream.

Experience with programming these classes of super-computer led to the development of new FORTRAN compilers (Perrott and Zarea-Aliabadi 1986). The Cray FORTRAN compiler took programs that were syntactically identical to standard FORTRAN and attempted to detect loops that could be vectorised. The DAP FORTRAN on the other hand allowed array parallel operations in APL style so that one could write:

V()+5	add 5 to the 64 elements of V
M.GT.0	generate a 4096 bit truth matrix
V(VI)	VI is an integer vector that indexes V
V(B)	B is a Boolean mask that selects V(I) where B(I) is true
IV(,5,6)/V	64 element vector IV is divided element by element by V

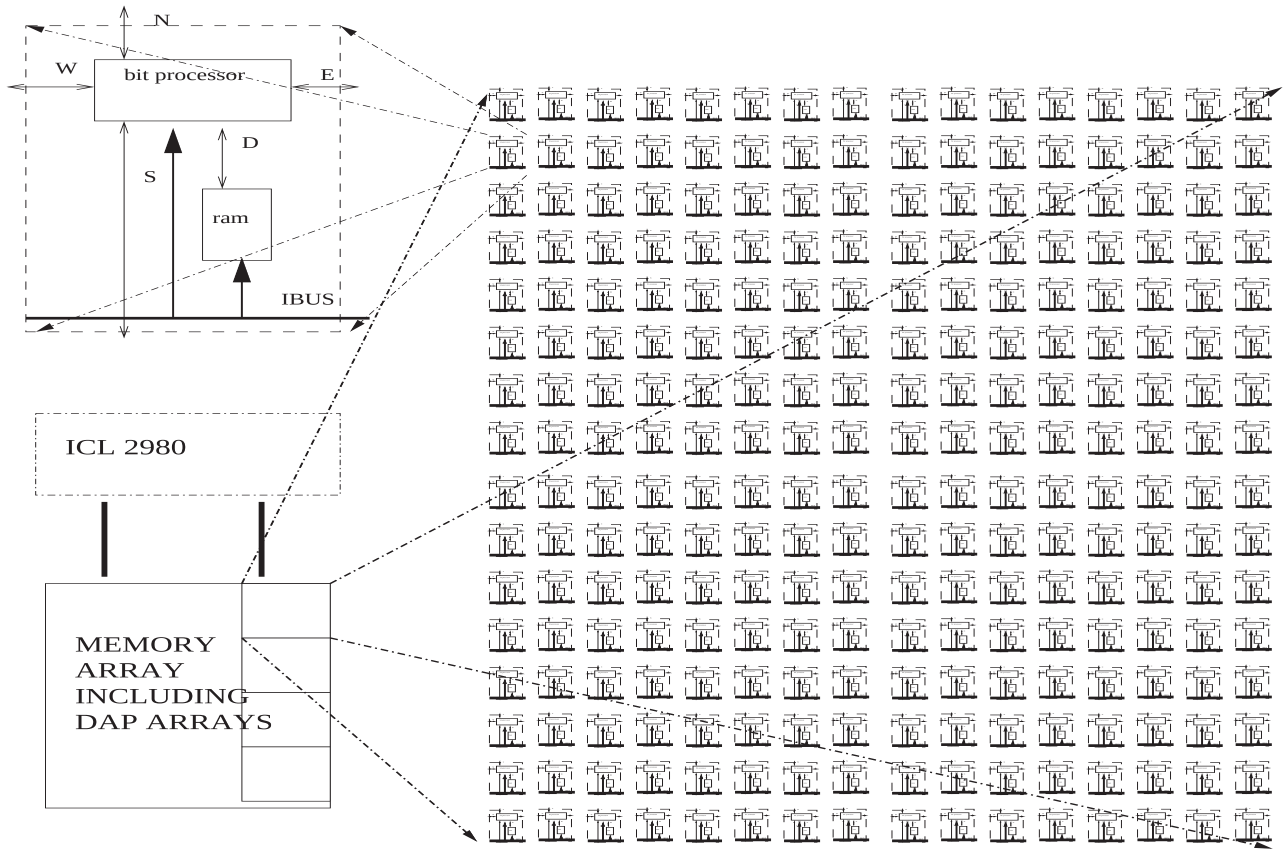

Figure 3: The ICL DAP showing a bit processor and 16th of the array of 4096 such processors

This sort of notation was later incorporated into standard Fortran90, High Performance FORTRAN (Ewing, Richardson, Simpson and Kulkarni 1998) and the Fortran90 subset F (Metcalf and Reid 1996).

Microprocessor based architectures

The next generation of super-computers tended to be shared memory multi-processors initially from makers like Cray but later from other manufacturers like Sun or IBM. The move to such machines was an effect of the production of high performance microprocessors. Large specialised vector machines of the earlier generation met limits in their performance in terms of clocking that could be bye-passed in an economical way through the use of replicated standard microprocessors. These in turn sparked off another round of array language development with languages like ZPL (Snyder 1999), Nesl (Blelloch 1995) and SAC (Grelck and Scholz 2003, Scholz 2003) borrowing APL concepts but being targeted at shared memory multi-processors. The aim in these languages was to split array operations over a group of processors, with all of the processors sharing the memory space within which the arrays were stored.

By the late 1980's microprocessors had reached an architectural complexity equivalent to the mainframe computers of an earlier generation. Faced with the problem of how to use an exponentially growing number of transistors, their designers started to copy techniques already pioneered in high performance mainframes. Initially this involved the introduction of super-scalar execution, but from the late 90s, SIMD and Vector processing features started to be incorporated. The Intel MMX architecture incorporated a modest sized SIMD co-processor on the die. With the P4 this was enlarged giving the architecture shown in Figure 4. There is a SIMD unit capable of operating on 16 byte wide operands. Alternatively, as with earlier SIMD designs, the byte processors can be ganged to form 8 x 16-bit processors, 4x32-bit processors or 2x64-bit processors. When ganged in 32-bit or 64-bit form, the SIMD processors can perform floating point as well as integer arithmetic. When operating with bytes they can perform saturated arithmetic which involves clipping the results on overflow to 255 or to 0 on underflow. This feature is particularly useful in image processing.

The significance of this development should be emphasised. What had once been an esoteric form of computer architecture, restricted to a few experimental super-computer sites was now the standard type of mass-produced PC. When operating on byte wide quantities, the P4 or Athlon architectures can speed their performance up by a factor of 10 or more by using the SIMD unit rather than the scalar processor. Using other bit widths the performance gains are appreciable but not as dramatic.

Software support for SIMD on PCs was initially poor. Intel's earlier C compilers allowed macro operations to operate on short vector types that corresponded to those supported directly in hardware. Later Intel released a FORTRAN compiler (Bik, Girkar, Grey and Tian 2002) that incorporated techniques earlier developed for Cray FORTRAN compilers to allow automatic SIMD vectorisation of loops in unmodified FORTRAN code. C compilers with similar features have been released by Intel and Codeplay. From the viewpoint of the Array Programming Language community such approaches fail to take full advantage of array architectures. The C and FORTRAN programs are still written with a sequential, operation at a time, mindset which can make it hard for programmers to think in parallel terms. If they have not cast their algorithm in parallel form to start out with, they may be unable to carry out the high level algorithmic transformations necessary to take full advantage of array architectures.

It was for this reason that the FORTRAN community incorporated APL concepts in the late 80s. Vector Pascal (Cockshott 2004, Cockshott 2002) is a Pascal extension analogous to High Performance Fortran that has been targeted at SIMD PCs. However, like HPF, it requires the programmer to explicitly allocate and declare arrays which is markedly less flexible than what occurs in APL.

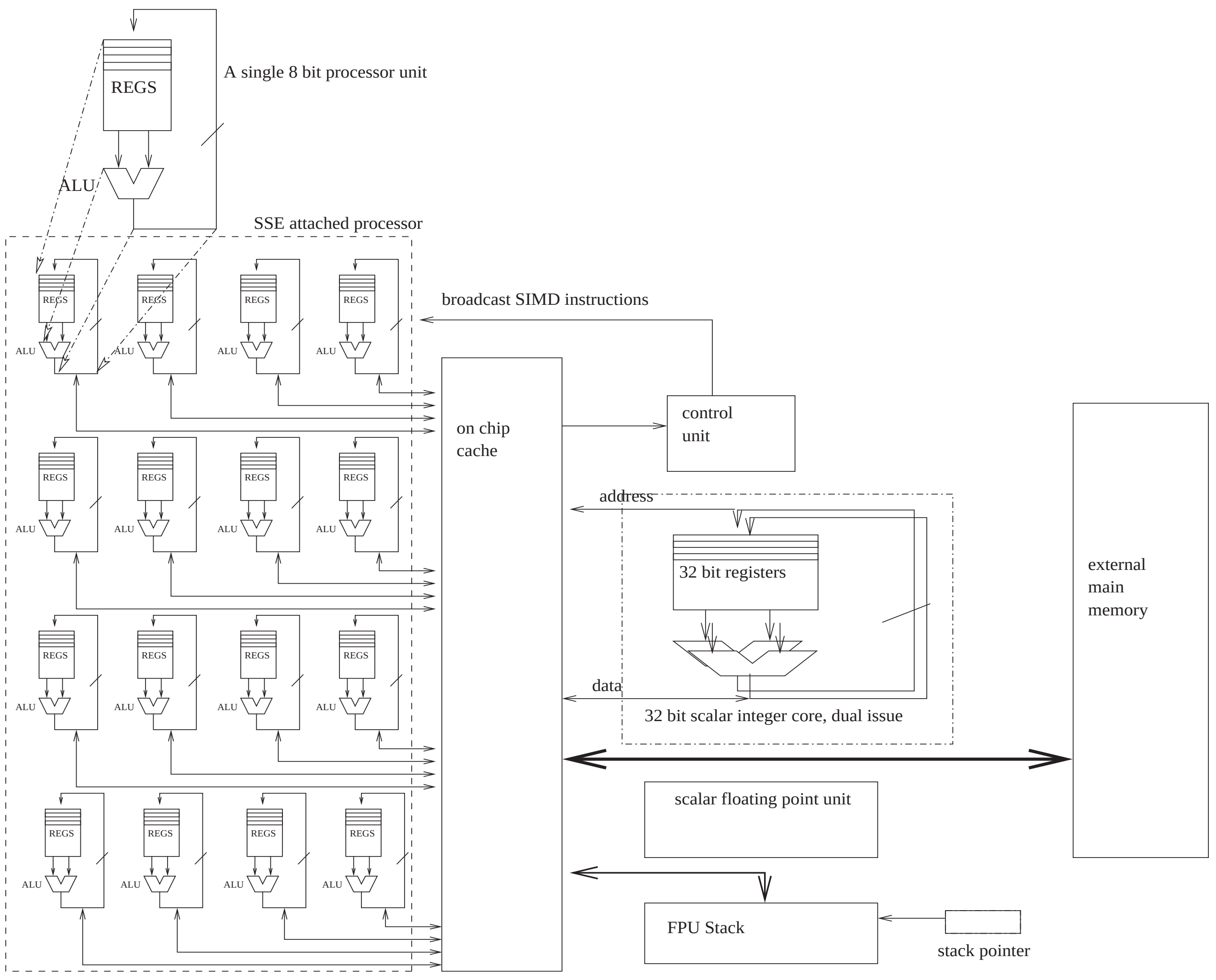

Figure 4: Outline of the Intel P4 architecture showing the incorporated Streaming SIMD unit.

Future machines

The SIMD unit of the P4 is targeted at graphics applications. The byte wide arithmetic is of obvious value in image manipulation. The ability to operate on 4 element vectors of floating point numbers is of particular use in 3D graphics where one performs viewpoint projection by multiplying 4 element vectors in homogenous co-ordinates by 4□4 rotation, translation and scale matrices. A computer game may require tens of thousands of vertices to be transformed in this way for every video frame - at say 25Hz. The demand for more realistic games has led chip designers to raise the level of parallelism even further. The pioneer in this was Sony, whose Emotion Engine, incorporated in the £100 Play Station 2 (PS2) included:

1. A MIPS 64 bit core acting as the control processor.

2. An integer SIMD unit analogous to that on the P4.

3. A pair of Vector Processing Units (VPU), each of which could perform 4x32-bit floating point operations each clock cycle.

The PS/2 differed from the machines discussed earlier by incorporating both SIMD and distributed memory multi-processing. The VPUs can operate as independent processors fetching their own instructions. They have vector floating point registers and also scalar registers for address manipulation. Instead of cache, they each have a modest sized private high speed memory on chip. Annoyingly they are asymmetrical in this with one having 16K and the other 64K of memory. Vector processing programs and data reside in this memory. DMA channels allow the MIPS core processor to transfer data to and from the private VPU memory and to initiate jobs on the VPUs.

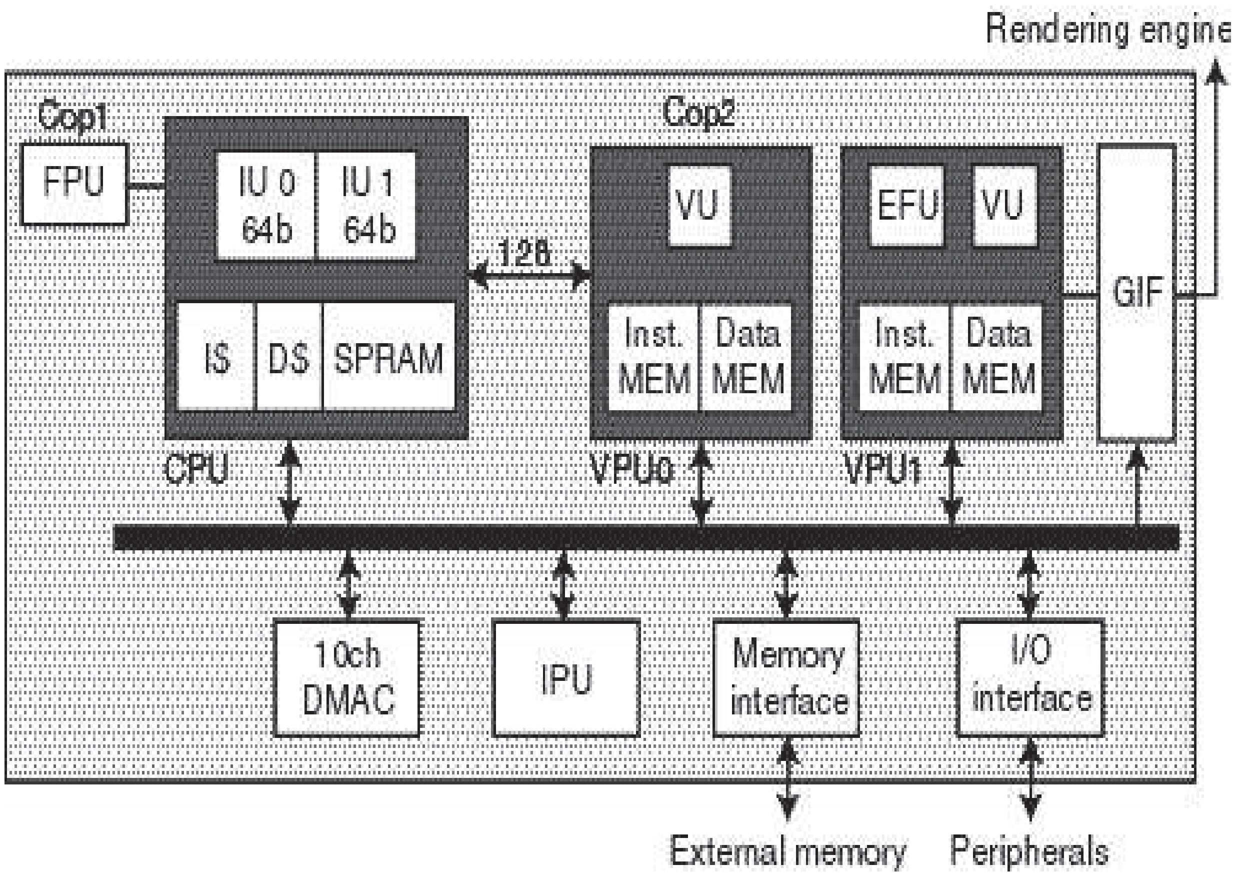

Figure 5: The architecture of the PS/2 processor. The EE core incorporates a SIMD unit analogous to that shown in Figure 4.

The Cell processor jointly developed by IBM and Sony for use in the PS/3 among other applications extends the model of the PS/2 by having 7 vector units each with 256K of memory on chip. These promise to be able to perform of the order of 50 Gigaflops provided that the VPUs can be kept busy. Similar architectures are being developed in other projects (Paulson 2005).

At the crux of making supercomputer-on-a-chip systems effective will be the programming necessary to get the most out of the circuitry and architecture. However, there are few programming tools and no programming languages optimized for this purpose (Paulson 2005).

By dint of great ingenuity it has been possible for vectorising compilers for scalar languages to detect a certain amount of parallelism. The new generation of processors raises the bar enormously. Not only must parallelism be detected, but data must be distributed among a heterogeneous collection of processors with private memories and differing instruction sets.

It seems plausible that only array languages which allow programmers to specify bulk operations in a data-parallel style, and which have full control over the allocation and placement of store will be able to meet these processing needs.

It is a moot point whether this implies compiled languages like SAC or interpretive languages like APL or J. Given the number of VPU's available to do the bulk calculations, it is plausible that an interpreter running on the control processor could dispatch jobs to the VPUs at a sufficient rate to keep them busy on large problems. In either case, array languages offer one of the most promising avenues for the massive parallelism offered by the new parallel vector machines. But implementing efficient array language implementations on these machines will demand some of the most sophisticated compilers and interpreters so far developed.[1]

[1] This paper was submitted for publication on October 11, 2005.

References

Abrams, P.: 1970, An APL Machine, Stanford Linear Accelerator Center, Stanford University, Stanford.

Bik, A. J. C., Girkar, M., Grey, P. M. and Tian, X.: 2002, Automatic intra-register vectorization for the intel architecture, Int. J. Parallel Program. 30(2), 65–98.

Blelloch, G.: 1995, Nesl: A nested data-parallel language, Vol. CMU-CS-95-170, Carnegie Mellon University.

Cockshott, P.: 2002, Vector pascal reference manual, SIGPLAN Not. 37(6), 59–81.

Cockshott, P.: 2004, Efficient compilation of array expressions, SIGAPL APL Quote Quad 34(2), 16–25.

Emerson W. Pugh, L. R. J. and Palmer, J. H.: 1991, IBM's 360 and Early 370 Systems, MIT.

Ewing, A., Richardson, H., Simpson, A. and Kulkarni, R.: 1998, Writing Data Parallel Programs with High Performance Fortran, Edinburgh ParallelComputing Centre.

Grelck, C. and Scholz, S.-B.: 2003, SAC — From High-level Programming with Arrays to Efficient Parallel Execution, Parallel Processing Letters 13(3), 401–412.

Hakami, B.: 1975, Efficient Implementation of APL in a Multilanguage Environment, International Computers Ltd.

Hassitt, A., Lageshulte, J. and Lyon, L.: 1973, Implementation of a high level language machine, Communications of the ACM 16.

Hillis, W. D.: 1986, The connection machine, MIT Press, Cambridge, MA, USA.

Lavington, S.: 1980, Early British Computers, Manchester University Press, Manchester.

Lonergan, W. and King, P.: 1985, Design of the b5000 system, in D. Sieworek, G. Bell and A. Newell (eds), Computer Structures, McGraw Hill.

Metcalf, M. and Reid, J.: 1996, The F Programming Language, Oxford Univesity Press.

Morse, S., Ravenel, B., Mazor, S. and Pohlman, W.: 1985, Intel microprocessors: 8008 to 8086, in D. Sieworek, G. Bell and A. Newell (eds), Computer Structures, McGraw Hill.

Naur, P. and Backus, J.: 1960, Report on the algorithmic language algol 60, Danish Academy of Technical Sciences, Copenhagen.

Palmer, J. and Morse, S.: 1984, The 8087 primer, Wiley, New York.

Paulson, L. D.: 2005, Squeezing supercomputers onto a chip, Computer 38(1), 21–23.

Perrott, R. H. and Zarea-Aliabadi, A.: 1986, Supercomputer languages, ACM Comput. Surv. 18(1), 5–22.

Reddaway, S. F.: 1973, Dap a distributed array processor, SIGARCH Comput. Archit. News 2(4), 61–65.

Russell, R.: 1985, The cray-1 computer system, in D. Sieworek, G. Bell and A. Newell (eds), Computer Structures, McGraw Hill.

Scholz, S.-B.: 2003, Single Assignment C — Efficient Support for High-Level Array Operations in a Functional Setting, Journal of Functional Programming 13(6), 1005–1059.

Smith, W., Rice, R., Cheskey, G., Laliotis, T. and Lundstrum, S.: 1985, The symbol computer, in D. Sieworek, G. Bell and A. Newell (eds), Computer Structures, McGraw Hill.

Snyder, L.: 1999, A Programmer's Guide to ZPL, MIT Press, Cambridge, Mass.

Turing, A.: 1937, On computable numbers, with an application to the Entscheidungsproblem, Proceedings of the London Mathematical Society 42, 230–65.