Processing In Memory: Chips to Petaflops

Peter M. Kogge
Jay B. Brockman
University of Notre Dame
Notre Dame, IN
{kogge, jbb}@cse.nd.edu

Thomas Sterling
California Institute of Technology
Pasadena, CA
tron@cacr.caltech.edu

Guang Gao
University of Delaware
Newark, DE
ggao@eecis.udel.edu

Abstract

This paper discusses the potential use of Processing-In-Memory (PIM) Technology in petaflops level computing. It starts with a quick review of a proposed PIM architecture called Shamrock, and follows that up with a discussion of several execution models that the architecture supports. Sizings for a petaflops-level machine constructed solely from PIM devices at several points in time are given. This is then projected to how PIM architectures will play a pivotal role in the recently initiated HTMT (Hybrid Technology Multi-Threaded) petaflops system architecture project.

Introduction

A peta(fl)ops is 10^{15} operations per second, which is a thousand times faster than the fastest computer demonstrated to date. While the advent of petaflops-level computing would revolutionize a broad range of application areas, the pursuit of a petaflops itself calls for revolutionary ideas in the areas of algorithms, software, and architectures.

To place the problem in perspective, we consider the development of a petaflops system using conventional technologies. Fig. 1 is a roadmap showing the memory and CPU chip count needed to realize a petaflops-level system, assuming one byte of memory per op, for a total of one petabyte. Based on SIA projections [2], to build such a system in 1998 would require over 30 million chips, of which 99 percent would be memory. By 2010, however, according to SIA projections, "only" 200K chips would be needed, with two memory chips per CPU. This in turn gives rise to a number of problems. First, hiding the latencies in such a system requires a high degree of multithreading and hence parallelism. In addition, the high-performance processors assumed at the time (16-way issue, 1.1 GHz) would require huge bandwidth from few memory parts.

Paradoxically, increasing memory density reduces the number of memory parts required to achieve a given memory size, which makes the bandwidth problem between CPUs and memory worse. For example, using a "gigaflops to the 3/4 power" rule [20] (typical of 3D plus time simulation problems) results in 2010 systems with 1 DRAM part for approximately 15 CPUs. To say that this requires a near-perfect cache hierarchy is an understatement.

Processing-In-Memory (PIM) technology combines on a single CMOS chip both logic and memory. This simple trick has a profound impact on computer architecture: the CPUs are much closer electrically to the memory arrays containing instructions and data, and the number of bits available from each access can be literally orders of magnitude greater than what one can transfer in a single cycle from today's conventional memory chip to today's conventional (and separate) CPU chip or cache system. Together, this greatly reduces memory latency and greatly increases memory bandwidth—the twin demons of modern computer design.

The result is that PIM gives the potential for doing away with much of the expensive memory hierarchy present in modern design, and replacing the CPU cores with simpler designs. It also allows for additional high performance techniques such as vector or SIMD processing to be placed inexpensively directly next to the memories where full use of the local bandwidth can be achieved. The result is less power, less silicon, and less complexity—which then allows us to place multiple such nodes on the same chip—providing even more performance per square of silicon by eliminating the need for complex inter-CPU communications paths. This is essential for petaflops levels of performance.

A series of papers, especially [3 and 17], summarize the state of PIM technology today [12, 13, 18]. Projections of what PIM-based high-performance computers might look like over the next twenty years are discussed in [9, 10, 11, 12, 20]. Estimates done for

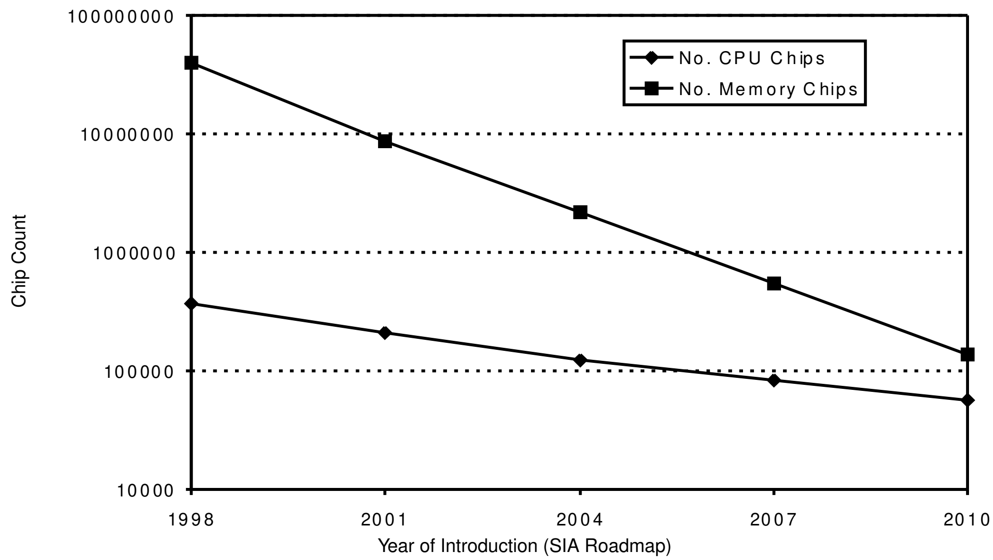

Figure 1. Petaflops sizing

the Frontiers' 96 conference [10] found that using 2004 technology, a 100 TF system with 1 TB of memory would utilize 5,800 PIM chips, each with 74 nodes per chip. In the same time frame a 100 TB system would employ 26,000 16 node chips. In either configuration each node would be capable of about 0.25 GF—considerably simpler than the potential CMOS workstation CPUs of the same era. In 2010 technology, a 1 TB system would employ 1,400 680-node chips, and a 100 TB system would employ 16,234 56-node chips, where each node now has a peak of about 1 GF.

Despite the gains possible by an all PIM petaflops-level machine, it still appears that such machines, even with the best of CMOS technologies, will require the programmer to deal with multi-million way parallelism. Many real applications may not permit such huge levels of parallelism. To attack such problems, a large scale collaboration among several research groups—the HTMT project [1, 5, 19]—is focusing on a mixed-technology solution where extremely long latencies are possible, and where "pre-emptive" activity in the PIM-based memory are essential to reduce or eliminate the latency penalties.

This paper takes one such proposed PIM architecture, Shamrock, and discusses a possible architecture for a petaflops machine based on it, along with execution and programming models to match such PIM components. The HTMT architecture is then introduced briefly, followed by a description of how specific PIM functions are essential to it.

Shamrock

Shamrock [8, 9, 10, 11], is the name that we have given to the result of a study into how to best "tile" the surface of a CMOS chip with repeating patterns of logic and memory areas that efficiently scale into a large, parallel computing chip. Fig. 2 illustrates the Shamrock floorplan. The basic premise is to start with a node that consists of logic for a CPU and data routing in the center, and four arrays of memory next to it, two on the top and two on the bottom. Each memory array is made up of multiple sub units, much as current memories are. However, the sense amplifiers for such memories, rather than face some multiplexing logic which strips away all but a handful of the bits at each access, face directly the CPU logic. This allows ALL the bits read from a memory in one cycle to be made available at the same time to the CPU logic.

Also key is that we have two separately addressable memories on each face of the CPU. When we tile the surface of a chip with such nodes, we arrange them in rows, but stagger alternating rows so that the two memory arrays on one side of one node impinge directly on the memory arrays of two DIFFERENT nodes in the next row. The result is a structure where each CPU has a true shared memory interface with four other CPUs, with the interface at the full width of the memory. Finally, all of this is done without expensive

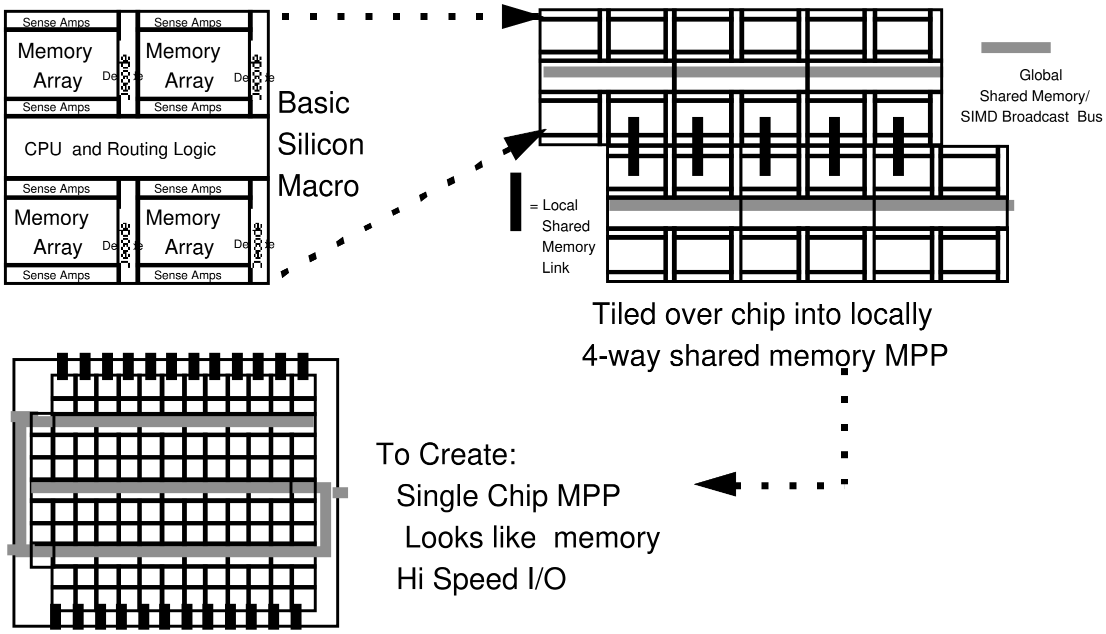

Figure 2. Shamrock floorplan

and space consuming global wires.

As shown in the chip floorplan in the lower-left corner of Fig. 2, off-chip connections come in two forms, both of which enhance scalability. First, peripheral contacts abut the top and bottom edges of the array of tiled nodes, from which connections can be made with adjoining chips in a way that maintains the tiled topology. Second, down the center of each logic strip is a global memory bus, with peripheral contacts on the left and right sides of the chip, that allows a processor on the outside to view the chip as "memory" in the conventional sense. Again, both of these connections stem naturally from the topology of the individual nodes, and do not require any expensive additional wiring on the chip.

The HTMT System

The Hybrid Technology Multi Threaded (HTMT) project [1, 5, 19] is a collaborative project among about half a dozen research groups (Cal Tech/JPL, U. Delaware, SUNY Stonybrook, Notre Dame, Princeton, plus an association with many other government and industrial labs) to define a system that can reach a petaflops level of performance in significantly less time than projected CMOS trends would allow. The current HTMT baseline attempts to avoid the multi-million way parallelism problem by a mix of technologies starting with perhaps 10,000 CPUs constructed from RSFQ (Rapid Single Flux Quantum) superconducting technology running at several hundred GHz, each with a small amount of local cryogenic RAM (CRAM). Because such technologies are inadequate for the system memory densities needed, HTMT has included two layers of memory above these devices: an SRAM and a DRAM layer, as shown in Fig. 3. Interconnecting the two are high speed optical networks. The DRAM layer then connects to other memory subsystems, including a 3D holographic storage subsystem and a huge disk farm.

In such a system, memory latency from the RSFQ CPUs down to the lower levels of the memory hierarchy becomes even more paramount than today, with cache miss penalties of hundreds of thousands of cycles possible if the system is architected conventionally. Instead, adopting PIM technology at both the SRAM and DRAM levels of the hierarchy allows systems where the memory takes pre-emptive action to prevent misses from occurring.

General PIM Execution Models

An execution model for a machine is a view of how programs execute on that machine. For Shamrock-like PIMs at a high level there are at least three such models that have been investigated, including: 1) an "accelerator model" where the PIMs act as a memory-resident "coprocessor" to one or more conventional

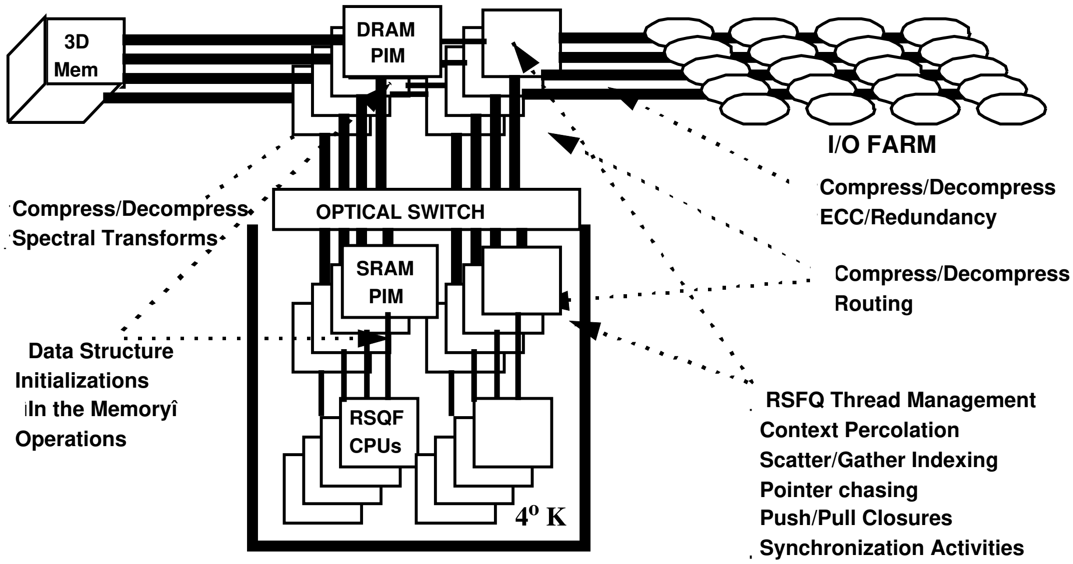

Figure 3. HTMT architecture

CPUs, 2) a "massively parallel peer" model where there is no host, only PIM chips, 3) a latency enhancing "active memory management" model where the PIMs are responsible for explicit management of the memory hierarchy of a supercomputer. The first two of these models are described [8]; the active memory model is most relevant to HTMT and is discussed below.

The Active Memory Model (AMM) represents a radical departure from both conventional models and the other two PIM models, and directly addresses the issues of run away memory latencies in very high speed systems such as HTMT. This model assumes a "conventional" arrangement of main memory connected through potentially several layers of intermediate storage to very high speed CPUs, such as RSFQ superconducting devices.

In such systems a cache miss from the lowest levels may result in miss penalties of literally hundreds of thousands of machine cycles. For obvious reasons we really want this to NEVER HAPPEN. The AMM model attacks this by assuming that the main memory, and potentially lower levels of memory in the hierarchy, are PIM-enhanced. This local processing capability then allows two capabilities which directly attack the latency problems. First, as in the accelerator model, a wide variety of functions can be done directly IN THE MEMORY, and never have to leave the memory for a tortuous trip to the main CPU. Such activities as initializing and performing simple vector operations on matrices all fall in this category. More interesting, memory local functions can also enhance the I/O capabilities of such a system. Compression, decompression, graphics generation, and the like can be performed on data directly as it is rolling between memory and I/O devices. This can hugely increase the apparent bandwidth (and storage density) of the I/O devices, without involving RSFQ CPU functions at all.

A second, and more direct, approach to latency reduction is to allow the PIM-enhanced memory to respond to a cache miss, or "prefetch instruction" from the main CPUs in more complex ways. Complex gather/scatter operations can respond to one request with the collection of data from all over memory into a single contiguous unit which fits within the lower levels of the memory hierarchy and enhances locality of accesses to them. (A striking example of this can be found in the SMC functions described in [16]).

Further, if two or more levels of the memory hierarchy are also PIM-enhanced, then it may also pay to employ compression and decompression in the transfer of data between levels of the hierarchy, enlarging the apparent bandwidth of the connection, and thus reducing again the apparent latency of the memory.

Finally, in the ultimate expression of latency reduction, the PIM-enhanced memories can do more than simply respond to requests. They can perform "preemptive strikes" which gather data (and code) together in contiguous packets for transmission down the hierarchy IN ADVANCE of requests from the main CPUs. In a sense this allows the PIMs to manage the transfer of data between spaces independently from the executing programs. A variety of potential programming models to achieve this split have been identified and are discussed below.

General PIM Programming Models

There are also multiple ways in which a programmer can approach the process of developing code for a PIM-based computer. Some of these map well onto only a single execution model. Others are suitable for several such models, and may in fact be invoked within the same program. Each is discussed briefly below.

Static Library Model

In this model, program access to the PIM is through a library of pre written functions. Arguments include pointers to the memory areas holding the operands, and the memory areas where the results should go. This is a good match for either the accelerator or the AMM model of execution, and allows virtually any programming language to access PIM functions without special compilation techniques. Relatively small runtimes are needed in each node (just enough to initialize a library function and handle exceptions), and the libraries themselves can be designed for compactness. In the EXECUBE PIM [13] such a runtime support package was extremely small and highly efficient.

The only drawbacks are that functions are limited to those that are predefined and available in the library, that the programmer must usually be aware of any assumptions the library makes about object layout in memory, and that it is hard to get anywhere near peak performance if we are relying on a conventional scalar CPU to decide when and where to initiate functions.

SPMD Model

This is an extension of the library model where a compiler will identify opportunities for parallelism, such as between loop iterations, and replace said loop by a new function call which activates a new function stored in the PIM library. This new function is constructed from the loop body, and compiled into native PIM CPU code. This approach was actually successfully used to partially parallelize some Ada program for the EXECUBE PIM. The problems are as before: limitations to the scalar capability of the main CPU, and the additional problem of dynamically managing the library overlays, especially in a multiple process environment.

Modify ON Access (MONA) Model

The MONA approach is a new one just beginning investigation at Notre Dame. In this model, objects in memory that are to receive PIM processing are treated as "file"-like objects to which one can perform traditional "open," "close," "read," and "write" like operations. In addition, however, functions can be "pushed" onto an object in between an open and a read or write. These functions act somewhat like "filters" in streams - any access to the object involves these filters. For example, if an object is kept in memory in a compressed fashion, pushing a "decompress" onto the object before accessing it will convert it into a normal form. Likewise pushing a "compress" onto the write path will recompress data updates to the object. A "pop" operation can similarly remove a filter. In terms of PIM operation, identifying the filters in this way allows a wide variety of evaluation strategies to be employed. "Eager beaver" operations in the PIM could be invoked as soon as the filter was pushed. More elaborate filters may perform some "upfront" PIM processing, but invoke PIM library functions as needed to support the individual accesses. Prototypes of this approach are currently under construction using traditional page fault mechanisms on conventional workstations. With extensions, it appears quite likely to be usable in any of the execution models, but most especially in the AMM model where it allows us in a very early and simple way to "prespecify" processing that may want to be done on data to be transferred between different levels of the memory hierarchy.

Locally Shared/Globally Distributed Model

Clearly, all the techniques for software development that have been used for modern SMPs and distributed memory parallel processors can be used in the PIM context. Message passing libraries are an obvious candidate. The major obstacle is the memory density. Today and for the next one to two generations of memory technology, the PIMs are going to have less memory per operation than conventional wisdom proscribes, and the additional storage per node to hold a copy of a microkernel, MPI library, and such are probably too expensive. This, however, will change with increasing memory density. For example, a million nodes each requiring a megabyte for operating system and runtime results in a terabyte total. This is huge in a 1 TB system but inconsequential in a 100 TB one.

Split Execution Model

In terms of latency management, especially for systems such as the AMM model, another programming model may turn out to be very significant. In this model, the compiler breaks the program into two concurrently executing pieces. One performs the specified computations, but against a memory space that matches what can be implemented at the lower levels of the memory hierarchy. The other, however, represents a "skeleton" of the program, namely the data movement instructions minus the computational "leaves." This part is executed in the memory PIMs and assembles/disassembles computational "closures" that are shipped up and down the memory hierarchy. Synchronization occurs only at the "closure" level.

Although not implemented in its totality, such a model does have some strong correlations to modern practices. For example, in most modern RISC microprocessors, the compilers go out of their way to schedule the loads and stores of CPU registers and memory in ways that individual computational instructions will almost never block. The CPU registers are treated as a separate address space just as discussed above. Another example is in many attached signal and vector processors. In the IBM 3838 array processor, for example, the unit had internally several pipelined floating point units which could directly address are relatively large (for its time) block of SRAM storage [15]. A separately programmable unit sat between this memory and the main DRAM memory. Its entire purpose was to manage data transfers between the two memories. Approximately a half dozen data transfer routines matched the transfer patterns for virtually all the vector and array functions implemented in the computation unit. Explicit synchronization instructions at the end of large sequences of computation and parallel data transfers kept everything in order, with very high efficiency.

Current efforts are under way to investigate automated ways to develop these data transfer programs. One example is the use of combinators [14] to represent the data movements and conventional operations for the computations. Programs expressed in such terms represent trees with the computational operations at the leaves, and the internal nodes as combinators. When executed in parallel, the combinators simply "reorder" their operands to position the correct values for the appropriate operations. A parallel simulator is approaching completion, and will be used initially to explore the efficiency of such an approach to several standard parallelizable operations such as matrix multiply.

PIM in the HTMT Program Execution Architecture Model

The HTMT architecture is based on a multithreaded program execution model which hides latency by context switching among concurrent threads. A necessary condition for a thread to become enabled is that all data required by the thread have been produced, and all control dependences are satisfied. However, one unique feature of the HTMT thread activation model is the introduction of an additional necessary condition for a thread to become enabled: it must also meet all "locality requirements" for efficient execution. Intuitively, all data and code referenced by the thread should become local before a thread can begin execution.

To realize the HTMT thread program execution model, a new multi-level multithreaded context management strategy, called thread percolation, is currently under study. This methodology and the underlying mechanisms it requires can be considered to be a combination of multithreading with dynamic prefetching of coarse-grain contexts. Prefetching in the past has concentrated on moving blocks of data within the memory hierarchy. The dynamics of the multi-level context management strategy cause "hot" contexts to move up towards the processors and "cool" contexts to drift to lower, slower, higher-capacity memory. The context percolation can be considered as a generalization and extension of some earlier schemes (e.g. the "register-cache" and "register use cache"), where percolation is now performed across the entire memory system [6, 7].

Control of this context percolation methodology requires smart memory with substantial control logic associated with the memory chips themselves—e.g. PIM. PIM functionality will be extended and tailored for use at the appropriate levels of the HTMT memory hierarchy to provide support for the HTMT memory model, including thread percolation. It will also include sophisticated in-memory operations, pointer tracing, compound atomic operations for synchronization and mutual exclusion.

Acknowledgements

This work has been supported in part by NASA grant NAG5-2998, National Science Foundation NSF ACS-9612028, and ACS-961210, as well as support from the NSA and DARPA. The authors would also like to recognize Andres Marquez and Kevin Theobald of the Computer Architecture and Parallel System Lab (CAPSL) of the U. of Delaware for their work in the HTMT project.

Summary

By alleviating bottlenecks in both latency and bandwidth between the CPU and memory, PIM technology holds great promise over conventional architectures in achieving petaflops-level computing. Nonetheless, a PIM-only system would still require on the order of a million or more way parallelism to reach a petaflops. The goals of the HTMT project are to both reduce the level of parallelism required to reach a petaflops and to manage it more efficiently, through a combination of 100 GHz superconducting RSFQ CPUs and multi-threaded program execution and architecture models.

Because memory may be 100,000 or more cycles away from the CPU in the HTMT architecture, it can no longer assume its traditional, passive role of providing data only when requested by the CPU. Rather, memory must take on an active role, anticipating the needs of the CPU and performing supporting computation. PIM, in effect, provides the capability to radically rethink the semantics of computer memory.

PIM is thus more than just another VLSI implementation technique. It will change computer architecture in a variety of creative ways that will have a significant impact on computer designs ranging from the smallest one chip systems to the largest systems we can foresee building in the next 15 years.

References

1. _________, Workshop Report: First Workshop on Hybrid Technology Multithreaded Architecture for Very High Performance Computing, Cal. Inst. of Tech., Feb. 25-26, 1997.

2. __________, The National Technology Roadmap for Semiconductors, Semiconductor Industry of America, San Jose, CA, 1994.

3. Brockman, J. B. and P. M. Kogge, "The Case for PIM," Notre Dame CSE Report TR 97-03, Jan. 1997, submitted to IEEE Computer, preliminary version at www.cse.nd.edu/tech_reports/

4. Dally, W., et al, "The Message Driven Processor: A Multicomputer Processing Node with Efficient Mechanisms," IEEE Micro, April 1992, pop. 23-38.

5. Gao, G., K. Likharev, P. Messina, T. Sterling, "Hybrid Technology Multithreaded Architecture," 6th Sump. on Frontiers of Massively Parallel Computation, Annapolis, MD, Oct. 25-31, 1996, pp. 98-105.

6. Hum, H. H., K. B. Theobald, and G. R. Gao, "Building Multithreaded Architectures with Off-the-Shelf Microprocessors," Proc. Int. Parallel Processing Symp., 1994, pp. 288-294.

7. Hum, H. H. and G. R. Gao, "A Novel High-Speed Memory Organization for Fine-Grain Multi-Thread Computing," Proc. of Parallel Architecture and Language Europe, 1991, pp. 34-51.

8. Kogge, P. M., and J. B. Brockman, V. Freeh, S. Bass, "Petaflops, Algorithms, and PIMs," 1997 Petaflops Algorithms Workshop, Williamsburg, VA, April 13-18, 1997.

9. Kogge. P. and R. Szczerba, Final Report: Scalable Spaceborne Computing Using PIM Technology, CSE TR 96-32, Univ. of Notre Dame, Notre Dame, IN, Nov., 1996.

10. Kogge, P., S. C. Bass, J. B. Brockman, D. Z. Chen, E, H. Sha, "Pursuing a Petaflop: Point designs for 100TF Computers Using PIM Technologies," 6th Symp. on Frontiers of Massively Parallel Computation, Annapolis, MD, Oct. 25-31, 1996, pp. 88-97.

11. Kogge, P. M. Final Report: Processing-In-Memory (PIM) Based Architectures for Petaflops Potential Massively Parallel Processing, Notre Dame CSE TR 96-25, Sept. 1996

12. Kogge, P. M., T. Sunaga, E. Retter, et al, "Combined DRAM and Logic Chip for Massively Parallel Applications," 16th IEEE Conf. on Advanced Research in VLSI, Raleigh, NC, IEEE Computer Society Press # PR07047, March 1995, pp. 4-16

13. Kogge, P.M., "The EXECUBE Approach to Massively Parallel Processing," 1994 Int. Conf. on

Parallel Processing, Chicago, IL, August, 1994, pp. 77-84.

14. Kogge, P. M., The Architecture of Symbolic Computers, McGraw Hill, NY, NY, 1991.

15. Kogge, P. M., The Architecture of Pipelined Computers, McGraw Hill/Hemisphere Press, NY, NY, 1981.

16. McKee, S., W. Wulf and T. Landon, “Bounds on Memory Bandwidth in Streamed Computations,” Lecture Notes in Computer Science 966: Europar'95 Parallel Processing, Stockholm, Sweden, Aug. 1995, pp. 83-100.

17. Patterson, D., et al, “A Case for Intelligent RAM,” IEEE Micro, March/April 1997, pp. 34-44.

18. Sunaga, T, P. Kogge, et al, “A Parallel Processing Chip with Embedded DRAM Macros,” IEEE J. of Solid State Circuits, Oct. 1996, pp. 1556-1559.

19. Sterling, T., G. Gao, K. Likharev, P. Kogge, M. MacDonald, “Steps to Petaflops Computing: A Hybrid Technology Multithreaded Architecture,” NAECON, 1997.

20. Sterling, T. P. Messina, and P. Smith, Enabling Technologies for Petaflops Computing, MIT Press, 1995.