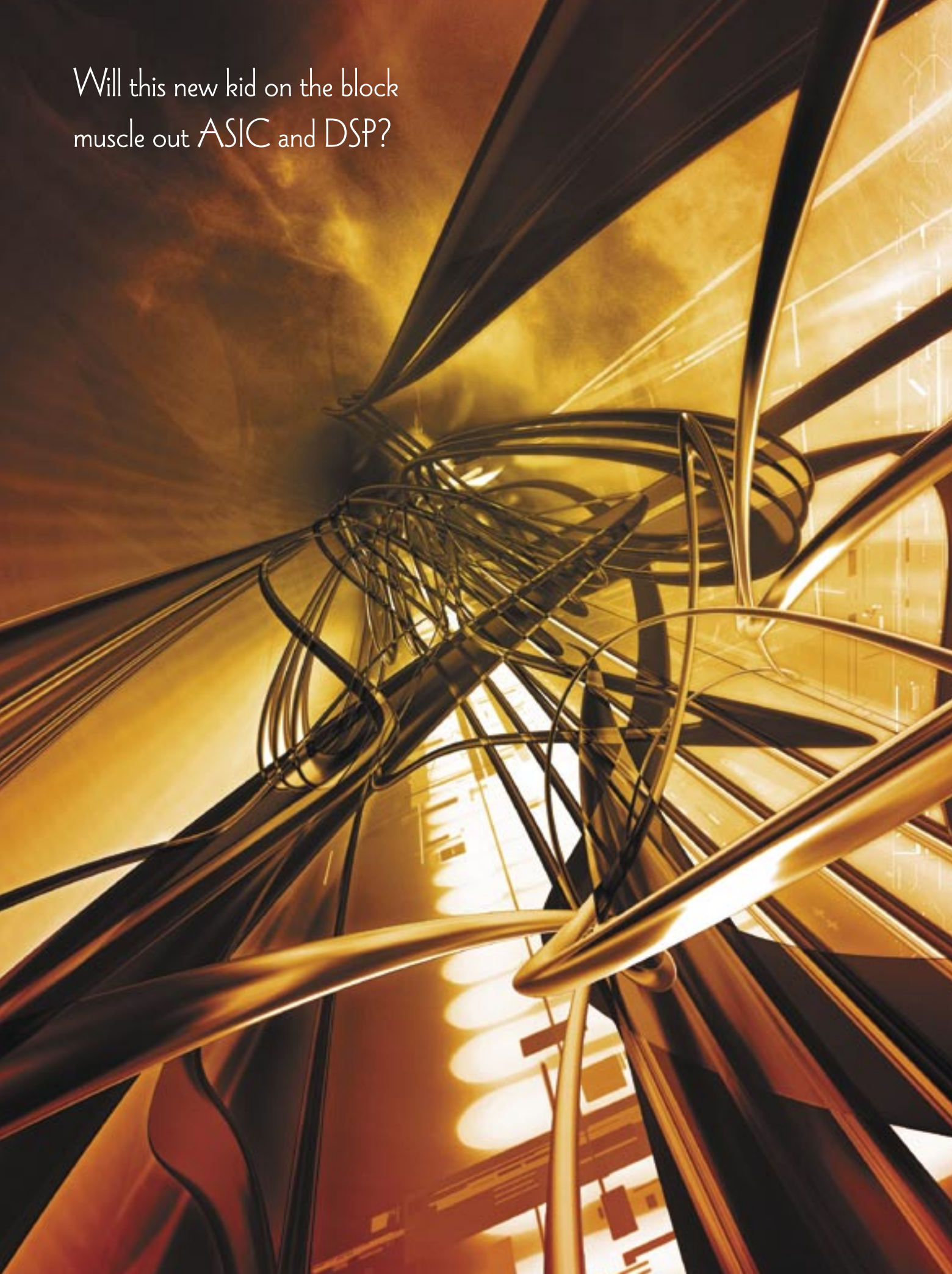
Stream Processors

Programmability with Efficiency

WILLIAM J. DALLY, UJVAL J. KAPASI, BRUCEK KHAILANY, JUNG HO AHN, AND ABHISHEK DAS, STANFORD UNIVERSITY

Many signal processing applications require both efficiency and programmability. Baseband signal processing in 3G cellular base stations, for example, requires hundreds of GOPS (giga, or billions, of operations per second) with a power budget of a few watts, an efficiency of about 100 GOPS/W (GOPS per watt), or 10 pJ/op (picoJoules per operation). At the same time programmability is needed to follow evolving standards, to support multiple air interfaces, and to dynamically provision processing resources over different air interfaces. Digital television, surveillance video processing, automated optical inspection, and mobile cameras, camcorders, and 3G cellular handsets have similar needs.

Conventional signal processing solutions can provide high efficiency or programmability, but are unable to provide both at the same time. In applications that demand efficiency, a hardwired application-specific processor—ASIC (application-specific integrated circuit) or ASSP (application-specific standard part)—has an efficiency of 50 to 500 GOPS/W, but offers little if any flexibility. At the other extreme, microprocessors and DSPs (digital


Will this new kid on the block
muscle out ASIC and DSP?

Stream Processors

Programmability with Efficiency

signal processors) are completely programmable but have efficiencies of less than 10 GOPS/W. DSP (digital signal processor) arrays and FPGAs (field-programmable gate arrays) offer higher performance than individual DSPs, but have roughly the same efficiency. Moreover, these solutions are difficult to program—requiring parallelization, partitioning, and, for FPGAs, hardware design.

Applications today must choose between efficiency and programmability. Where power budgets are tight, efficiency is the choice, and the signal processing is implemented with an ASIC or ASSP, giving up programmability. With wireless communications systems, for example, this means that only a single air interface can be supported or that a separate ASIC is needed for each air interface, with a static partitioning of resources (ASICs) between interfaces.

Stream processors are signal and image processors that offer both efficiency and programmability. Stream processors have efficiency comparable to ASICs (200 GOPS/W), while being programmable in a high-level language.

EXPOSING PARALLELISM AND LOCALITY

A stream program (sometimes called a synchronous data-flow program) expresses a computation as a signal flow graph with streams of records (the edges) flowing between computation kernels (the nodes). Most signal-processing applications are naturally expressed in this style. For example, figure 1 shows a stream program that performs stereo depth extraction based on the algorithm of Kanade.[1] In this application, a stereo pair of images are first filtered and then compared with each other to extract the depth at each pixel of the image. Along the top path, a *stream* of pixels from the left image is filtered by a Gaussian *kernel* to reject high-frequency noise, generating a stream of smoothed pixels. This stream is filtered by a Laplacian kernel to highlight edges, generating a stream of edge-enhanced pixels. The right image follows a similar filtering path. The SAD (sum of absolute differences) kernel then compares a 7x7 sub-image about each pixel in the filtered left image with a row of 7x7 sub-images in the filtered right image to find the best match. The position of the best match gives the disparity between the two images at that pixel, from which we can derive the depth of the pixel.

The stream program of figure 1 exposes both parallelism and locality. Each element of each input stream (all of the image pixels) can be processed simultaneously, exposing large amounts of data parallelism. This parallelism is particularly easy to identify in a stream program because the structure of the program makes the dependencies between kernels explicit. The complex disambiguation required when intermediate data is passed through memory arrays is not needed. Within each kernel, instruction-level parallelism is exposed since many independent operations can execute in parallel. Finally, the kernels can operate in parallel, operating on pixels or frames in a pipelined manner to expose thread-level parallelism.

The stream program also exposes two types of locality: *kernel* and *producer-consumer*. During the execution of a kernel, all references are to variables local to the kernel except for values read from the input stream(s) and written to the output stream(s). This is *kernel locality*. Consider

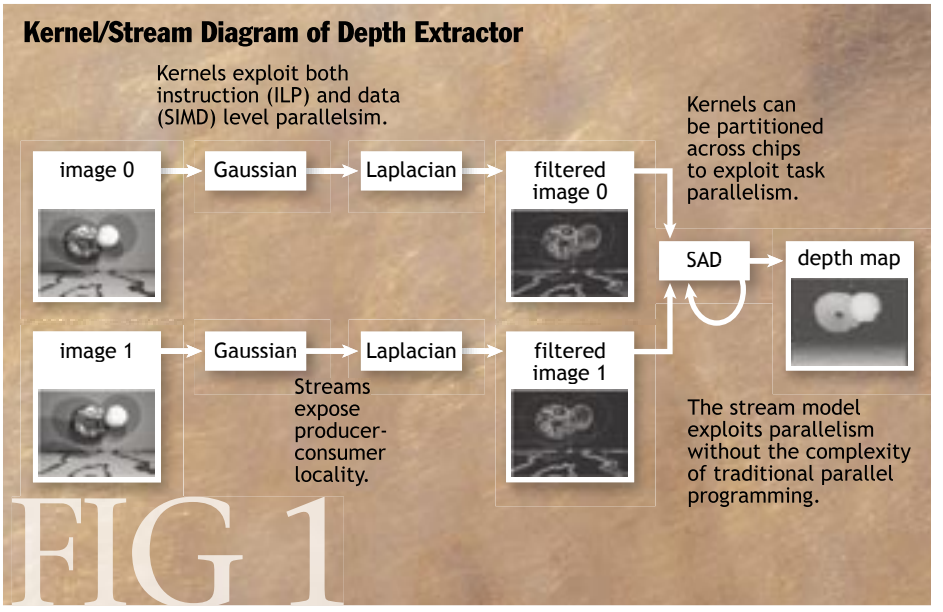


FIG 1

one implementation of the 7x7 convolution kernel. The inputs and outputs for the operations in the kernel require 176 references to values in local register files for every word accessed from the SRF (stream register file). Thus, because of kernel locality, we are able to ensure that one out of every 177 references is local to the kernel.

Producer-consumer locality is exposed by the streams flowing between kernels. As one kernel produces stream elements, the next kernel consumes these elements in sequence. By appropriately sequencing kernels, the elements of an intermediate stream can be kept local—values are consumed soon after they are produced. Each time the Gaussian kernel in figure 1 generates a block of the smoothed pixel stream, for example, this block can be consumed by the Laplacian kernel. Only a block of the intermediate stream exists at any point in time, and this block can be kept in local storage.

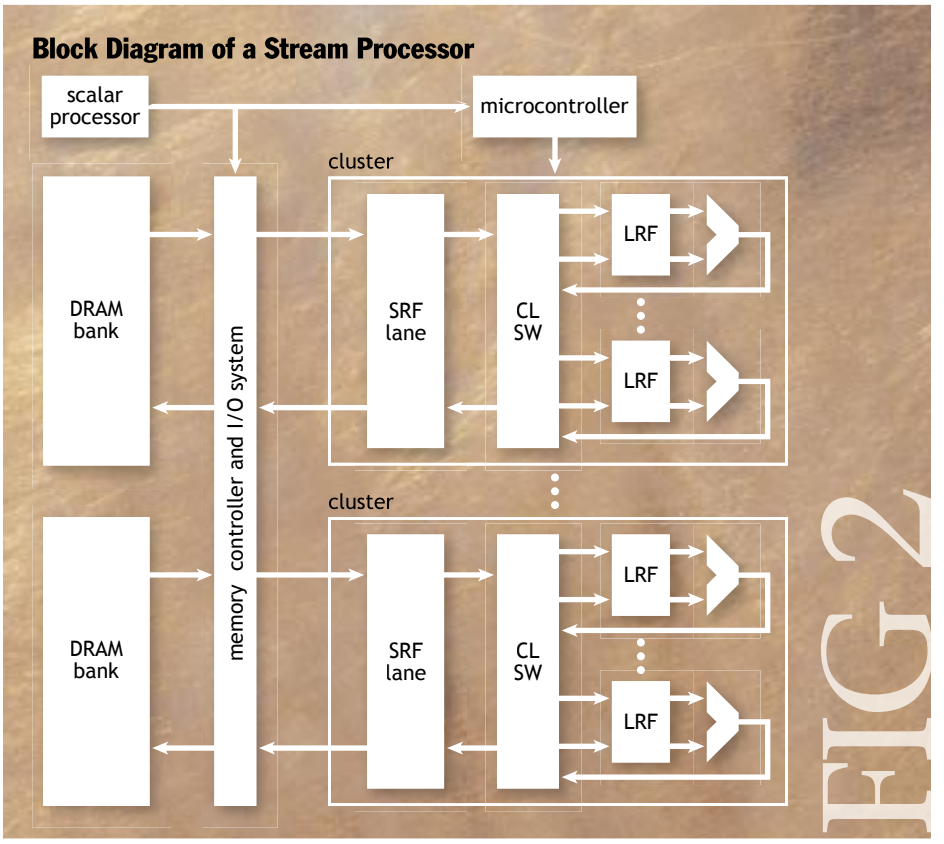
EXPLOITING PARALLELISM AND LOCALITY

As shown in the block diagram of figure 2, a stream processor consists of a scalar processor, a stream memory system, an I/O system, and a stream execution unit, which consists of a microcontroller and an array of C arithmetic *clusters*. Each cluster contains a portion of the SRF, a collection of A arithmetic units, a set of local register files, and a local switch. A local register file is associated with each arithmetic unit. A global switch allows the clusters to exchange data.

A stream processor executes an instruction set extended with kernel execution and stream load and store instructions. The scalar processor fetches all instructions. It executes scalar instructions itself, dispatches kernel execution instructions to the microcontroller and arithmetic clusters, and dispatches stream load and store instructions to the memory or I/O system. For each kernel execution instruction, the microcontroller starts execution of a microprogram broadcasting VLIW (very-long instruction word) instructions across the clusters until the kernel is completed for all records in the current block.

A large number, C×A, of arithmetic units in a stream processor exploit the parallelism of a stream program. A stream processor exploits data parallelism by operating on C stream elements in parallel, one on each cluster, under SIMD (single-instruction, multiple-data) control of the microcontroller. The instruction-level parallelism of a kernel is exploited by the multiple arithmetic units in each cluster that are controlled by the VLIW instructions issued by the microcontroller. If needed, thread-level parallelism can be exploited by operating multiple stream execution units in parallel. Research has shown that typical stream programs have sufficient data and instruction-level parallelism for media applications to keep more than 1,000 arithmetic units productively employed.[2]

The exposed register hierarchy of the stream processor exploits the locality of a stream program. Kernel locality is exploited by keeping almost all kernel variables in local register files immediately adjacent to the arithmetic units in which they are to be used. These local register files provide very high bandwidth and very low power for accessing local variables. Producer-consumer local-



Stream Processors

Programmability with Efficiency

ity is exploited via the SRF. A producing kernel, such as the Gaussian kernel in figure 1, generates a block of an intermediate stream into the SRF, each cluster writing to its local portion of the SRF. A consuming kernel, such as the Laplacian kernel in figure 1, then consumes the block of stream elements directly from the SRF.

To see how parallelism and locality are exploited in practice, figure 3 shows how the depth extraction program of figure 1 is mapped to the stream processor of figure 2. The input images are read from external memory or an I/O device into the SRF one block at a time, then the Gaussian kernel is run. Each cluster performs the kernel on a different pixel of the input, reading each pixel of the block from the SRF and writing each pixel of the output block to the SRF. Most local variables for the kernels are kept in the local register files with some partial products cycled through the SRF. Overall, for each word accessed from memory or I/O, 23 words are referenced from the SRF, and 317 are referenced from local registers.

This high fraction of references from local registers is not unique to the depth extractor. Figure 4 shows that kernel locality and producer-consumer locality exist in a broad range of applications and that a stream processor can successfully exploit this locality. The figure shows the bandwidth from main memory, the SRF, and local register files for six applications. The first column (depth) shows the bandwidth for the depth extractor of figure 1. MPEG is an MPEG2 encoder including motion estimation. QRD is a QR decomposition using the Householder method. STAP (space-time adaptive processing) is an adaptive beam-forming application. Render is an OpenGL 3D graphics-rendering program. RTSL (realtime shading language) is a renderer with a programmable shader.[3] For all of these programs, more than 95 percent of all references are from the local registers and less than 0.5 percent are from external memory.

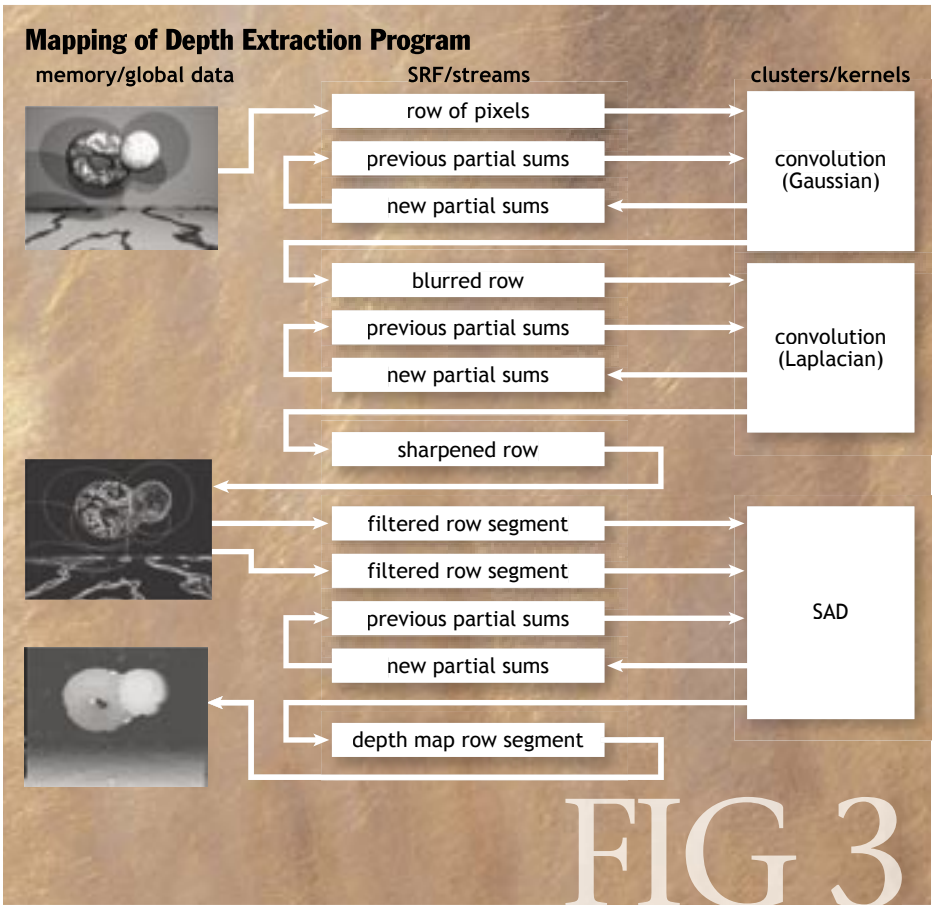


FIG 3

While a conventional microprocessor or DSP can benefit from the locality and parallelism exposed by a stream program, it is unable to fully realize the parallelism and locality of streaming. A conventional processor has only a few (typically fewer than four, compared with hundreds for a stream processor) arithmetic units and thus is unable to exploit much of the parallelism exposed by a stream program. A conventional processor is unable to realize much kernel locality because it has too few processor registers (typically fewer than 32, compared with thousands for a stream processor) to capture the working set of a kernel. A processor's cache memory is unable to exploit much

of the producer-consumer locality because there is little reuse of consumed data (the data is read once and discarded). Also, a cache is reactive, waiting for the data to be requested before fetching it. In contrast, data is proactively fetched into an SRF so it is ready when needed. Finally, a cache replaces data without regard to its liveness (using a least-recently used or random replacement strategy) and often discards data that is still needed. In contrast, an SRF is managed by a compiler in such a manner that only dead data (data that is no longer of interest) is replaced to make room for new data.

EFFICIENCY

Most of the energy consumed by a modern microprocessor or DSP is consumed by data and instruction movement, not by performing arithmetic. As illustrated in Table 1, for a 0.13µm (micrometer) process operating from a 1.2V supply, a simple 32-bit RISC processor consumes 500 pJ to perform an instruction,[4] whereas a single 32-bit arithmetic operation requires only 5 pJ. Only 1 percent of the energy consumed by the instruction is used to perform arithmetic. The remaining 99 percent goes to *overhead*. This overhead is divided between instruction overhead (reading the instruction from a cache, updating the program counter, instruction decode, transmitting the instruction through a series of pipeline registers, etc.) and data overhead (reading data from a cache, reading and writing a multiport register file, transmitting operands and intermediate results through pipeline registers and bypass multiplexers, etc.).

A stream processor exploits data and instruction locality to reduce this overhead so that approximately 30 percent of the energy is consumed by arithmetic operations. On the data side, the locality shown in figure 4 keeps most data movements over short wires, consuming little energy. The distributed register organization with a number of small local register files connected by a cluster switch is significantly more efficient than a single global register file.[5] Also, the SRF is accessed only once every 20 operations on average, compared with a data cache that is accessed once every three operations on average, greatly reducing memory access energy. On the instruction side, the energy required to read a microinstruction from the microcode memory is amortized across the data parallel clusters of a stream processor. Also, kernel microinstructions are simpler and hence have less control overhead than the RISC instructions executed by the scalar processor.

TABLE 1

Energy Per Operation (0.13µm, 1.2V)

Operation	Energy
32-bit arithmetic operation	5 pJ
32-bit register read	10 pJ
32-bit 8KB RAM read	50 pJ
32-bit traverse 10mm wire	100 pJ
Execute instruction	500 pJ

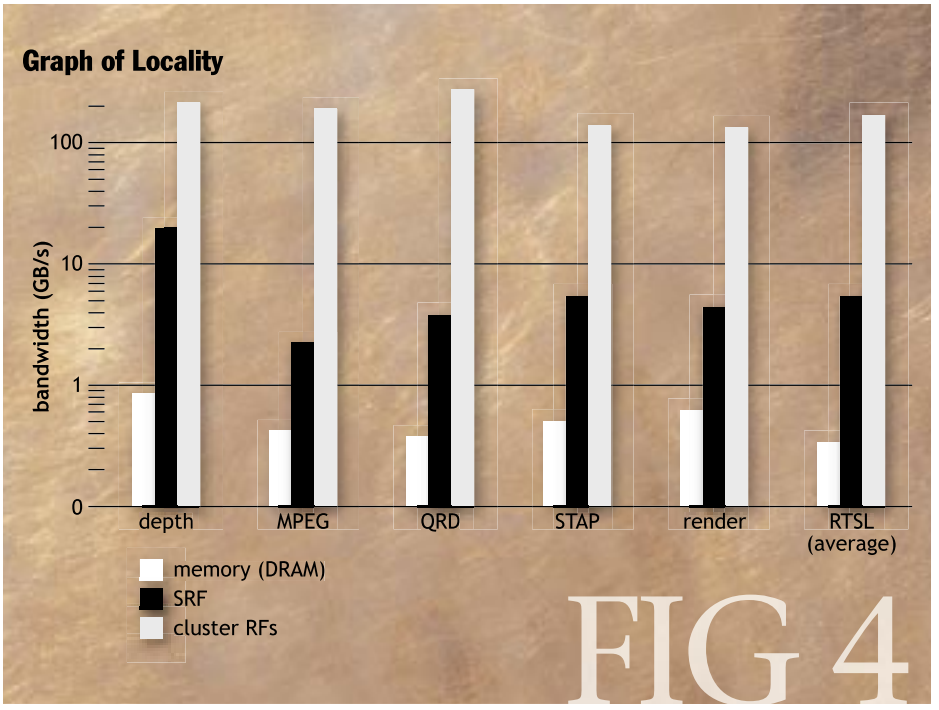


FIG 4

TIME VERSUS SPACE MULTIPLEXING

A stream processor time-multiplexes its hardware over the kernels of an application. All of the clusters work together on one kernel—each operating on different data—then they all proceed to the next kernel, and so on. This is shown on the left side of figure 5. In contrast, many tiled architectures (DSP

Stream Processors

Programmability with Efficiency

arrays) are space-multiplexed. Each kernel runs continuously on a different tile, processing the data stream in sequence, as shown on the right side of figure 5. The clusters of a stream processor exploit data parallelism, whereas the tiles of a DSP array exploit thread-level parallelism.

Time multiplexing has two significant advantages over space multiplexing: load balance and instruction efficiency. As shown in figure 5, with time multiplexing the load is perfectly balanced across the clusters—all of the clusters are busy all of the time. With space multiplexing, on the other hand, the tiles that perform shorter kernels are idle much of the time as they wait for the tile running the longest kernel to finish. The load is not balanced across the tiles: Tile 0 (the bottleneck tile) is busy all of the time, while the other tiles are idle much of the time. Particularly when kernel execution time is data dependent (as with many compression algorithms), load balancing a space-multiplexed architecture is impossible. A time-multiplexed architecture, on the other hand, is always perfectly balanced. This often results in a 2x to 3x improvement in efficiency.

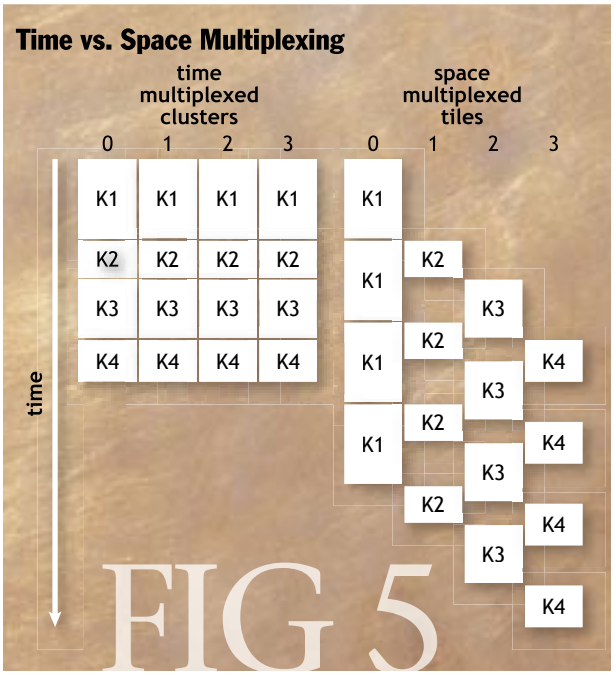


FIG 5

Exploiting data parallelism rather than thread-level parallelism, a time-multiplexed architecture uses its instruction bandwidth more efficiently. Fetching an instruction is costly in terms of energy. The instruction pointer is incremented, an instruction cache is accessed, and the instruction must be decoded. The energy required to perform these operations often exceeds the energy performed by the arithmetic carried out by the instruction. On a space-multiplexed architecture, each instruction is used exactly once, and thus this instruction cost is added directly to the cost of each instruction. On a time-multiplexed architecture, however, the energy cost of an instruction is amortized across the parallel clusters that all execute the same instruction in parallel. This results in an additional 2x to 3x improvement in efficiency.

STREAM PROGRAMMING TOOLS

Mapping an application to a stream processor involves two steps: *kernel scheduling*, in which the operations of each kernel are scheduled on the arithmetic units of a cluster; and *stream scheduling*, in which kernel executions and data transfers are scheduled to use the SRF efficiently and to maximize data locality. We have developed a set of programming tools that automate both of these tasks so that a stream processor can be programmed entirely in C without sacrificing efficiency.

Our kernel scheduler takes a kernel described in kernel C and compiles it to a VLIW microprogram. This compilation uses *communication scheduling*[6] to map each operation to a cycle number and arithmetic unit, and simultaneously schedule data movement necessary to provide operands. The compiler software pipelines inner loops, converting data parallelism to instruction-level parallelism where it is required to keep all operation units busy. To handle conditional (if-then-else) structures across the SIMD clusters, the compiler uses predication and conditional streams.[7] Figure 6 shows the schedule for the 7x7 convolution kernel from the depth extractor of figure 2 compiled to the Imagine stream processor (described later). Time is shown on the vertical axis and function units on the horizontal axis. The kernel scheduler is able to keep the multipliers (columns 4 and 5) busy nearly every cycle.

The stream scheduler schedules not only the transfers of blocks of streams between memory, I/O devices, and the SRF, but also the execution of kernels. This task is comparable to scheduling DMA (direct memory access) transfers between off-chip memory and I/O that must be

performed manually for most conventional DSPs. The stream scheduler accepts a C++ program and outputs machine code for the scalar processor including stream load and store, I/O, and kernel execution instructions. The stream scheduler optimizes the block size so that the largest possible streams are transferred and operated on at a time, without overflowing the capacity of the SRF. This optimization is similar to the use of cache blocking on conventional processors and to stripmining of loops on vector processors. Figure 7 shows a stream schedule for an OpenGL polygon renderer. Time is shown vertically and space in the SRF horizontally.

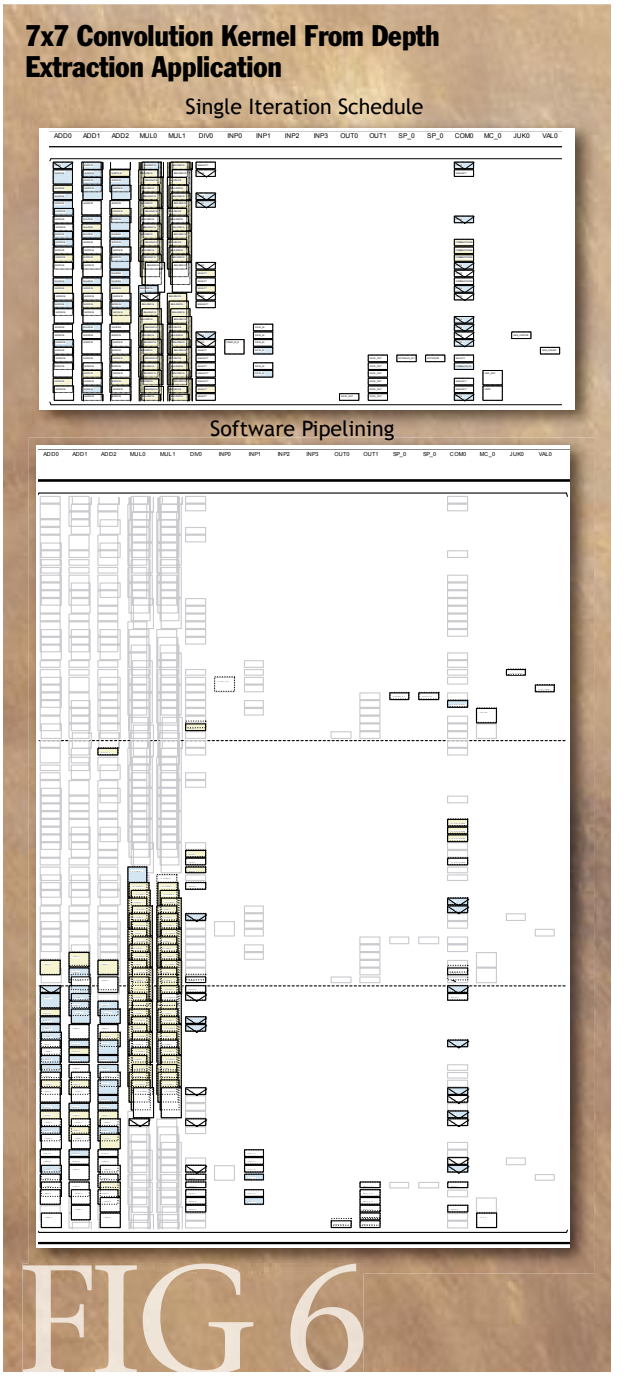
THE IMAGINE STREAM PROCESSOR

Imagine,[8] shown in figure 8, is a prototype stream processor fabricated in a 0.18µm CMOS process. Imagine contains eight arithmetic clusters, each with six 32-bit floating-point arithmetic units: three adders, two multipliers, and one divide-square root (DSQ) unit. With the exception of the DSQ unit, all units are fully pipelined and support 8-, 16-, and 32-bit integer operations, as well as 32-bit floating-point operations. Each input of each arithmetic unit has a separate local register file of sixteen or thirty-two 32-bit words. The SRF has a capacity of 32KB 32-bit words (128KB) and can read 16 words per cycle (two words per cluster). The clusters are controlled by a 576-bit microinstruction. The microcontrol store holds 2K such instructions. The memory system interfaces to four 32-bit-wide SDRAM banks and reorders memory references to optimize bandwidth. Imagine also includes a network interface and router for connection to I/O devices and to combine multiple Imagines for larger signal-processing tasks.

CHALLENGES

Stream processors depend on parallelism and locality for their efficiency. For an application to stream well, there must be sufficient parallel work to keep all of the arithmetic units in all of the clusters busy. The parallelism need not be regular, and the work performed on each stream element need not be of the same type or even the same amount. If there is not enough work to go around, however, many of the stream processor's resources will idle and efficiency will suffer. For this reason, stream processors cannot efficiently handle some control-intensive applications that are dominated by a single sequential thread of control with little data parallelism. A streaming application must also have sufficient kernel and producer-consumer locality to keep global bandwidth from becoming a bottleneck. A program that makes random memory references and does little work with each result fetched, for example, would be limited by global bandwidth and not benefit from streaming. Happily, most signal processing applications have adequate data parallelism and locality.

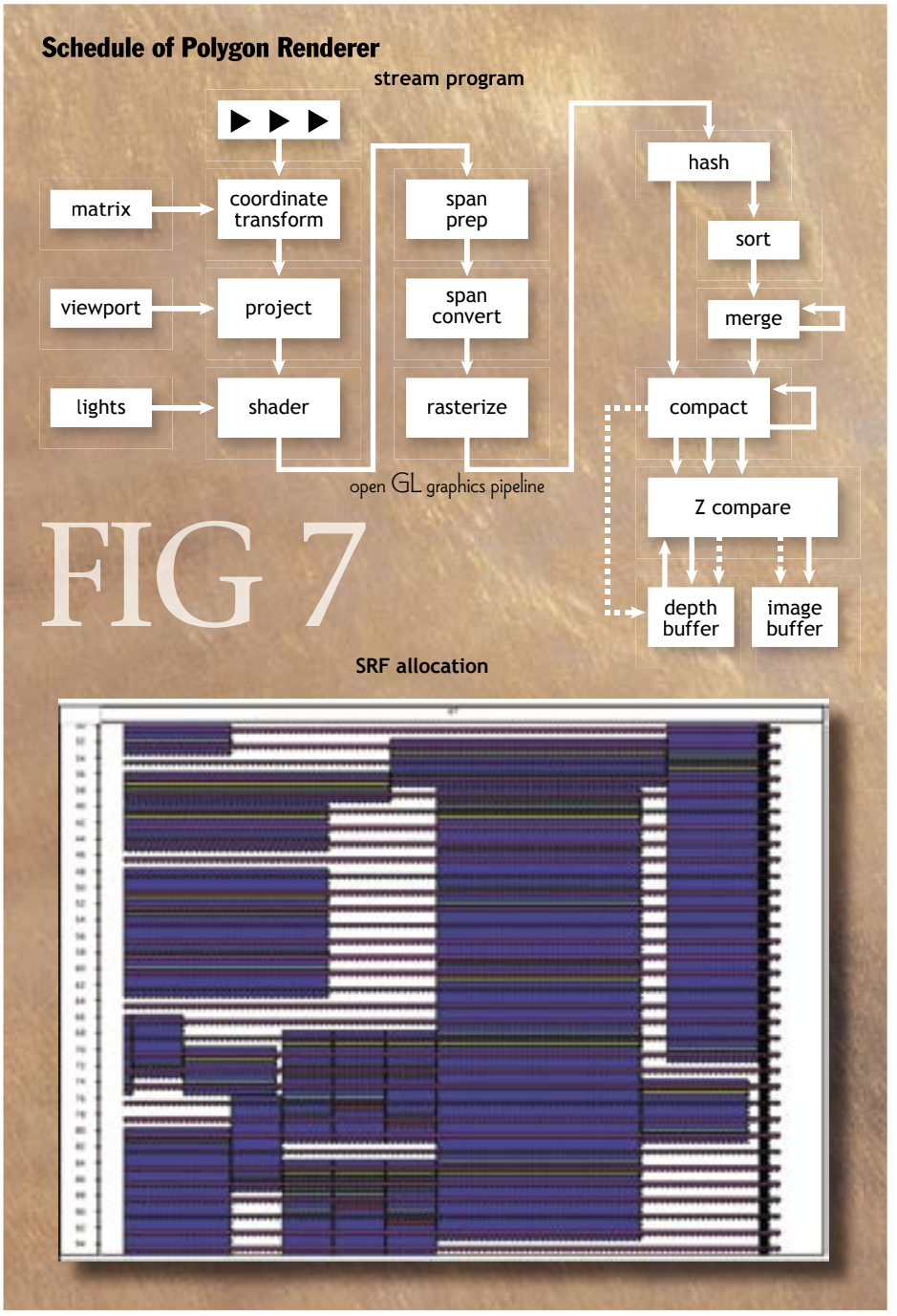
Even for those applications that do stream well, inertia represents a significant barrier to the adoption of stream processors. Though it is easy to program a stream proces-



Stream Processors

Programmability with Efficiency

sor in C, learning to use the stream programming tools and writing a complex streaming application still represents a significant effort. For evolutionary applications, it is often easier to reuse the existing code base for a conventional DSP, or the existing netlist for an ASIC rather than to develop new streaming code. An application must require both efficiency and flexibility to overcome this inertia.



THE FUTURE IS STREAMS

Figure 9 shows a roadmap that illustrates how we expect stream processors to evolve with improving semiconductor process technology. The figure shows two lines of evolution. The top line represents floating-point stream processors (that, like Imagine, support 32-bit floating-point operations), and the bottom line represents fixed-point processors that support just 8-, 16- and 32-bit integer operations. Integer operations are sufficient for most signal processing operations and, as the figure indicates, are significantly more efficient in terms of both area and power. Each point in the roadmap represents a stream processor (integer or floating point) implemented in a particular technology with a nominal die size of 1 cm^2 (1.3 cm^2 for the floating-point processors). For each point, the figure shows the performance and power at full voltage and the performance and power at reduced voltage (for more efficient operation). The performance, power, and area may be scaled up or down over a wide range by

varying the number of clusters in the stream processor.[9]

The main competition for stream processors are fixed-function (ASIC or ASSP) processors. Though ASICs have efficiency as good as or better than stream processors, they are costly to design and lack flexibility. It takes about $15 million and 18 months to design a high-performance signal-processing ASIC for each application, and this cost is increasing as semiconductor technology advances. In contrast, a single stream processor can be reused across many applications with no incremental design cost, and software for a typical application can be developed in about six months for about $4 million.[10] In addition, this flexibility improves efficiency in applications where multiple modes must be supported. The same resources can be reused across the modes, rather than requiring dedicated resources for each mode that remain idle when the system is operating in a different mode. Also, flexibility permits new algorithms and functions to be easily implemented. Often the performance and efficiency advantage of a new algorithm greatly outweighs the small advantage of an ASIC over a stream processor.

FPGAs are flexible, but lack efficiency and programmability. Because of the overhead of gate-level configurability, processors implemented with FPGAs have an efficiency of 2-10 MOPS per megawatt, comparable to that of conventional processors and DSPs. Newer FPGAs include large function blocks such as multipliers and microprocessors to partly address this efficiency issue. Also, though FPGAs are flexible, they are not programmable in a high-level language. Manual design to the register-transfer level is required for an FPGA, just as with an ASIC. Advanced compilers may someday ease the programming burden of FPGAs.

With competitive energy efficiency, lower recurring costs, and the advantages of flexibility, we expect stream processors to replace ASICs in the most demanding of signal-processing applications. Q

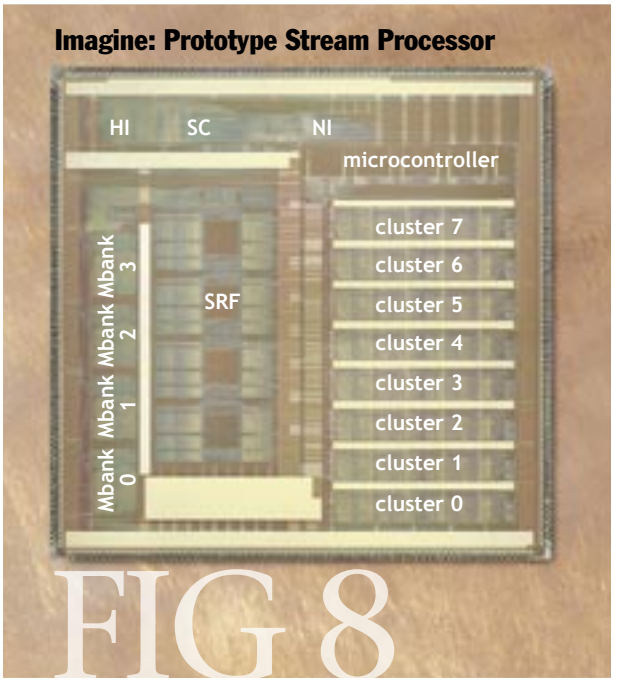


FIG 8

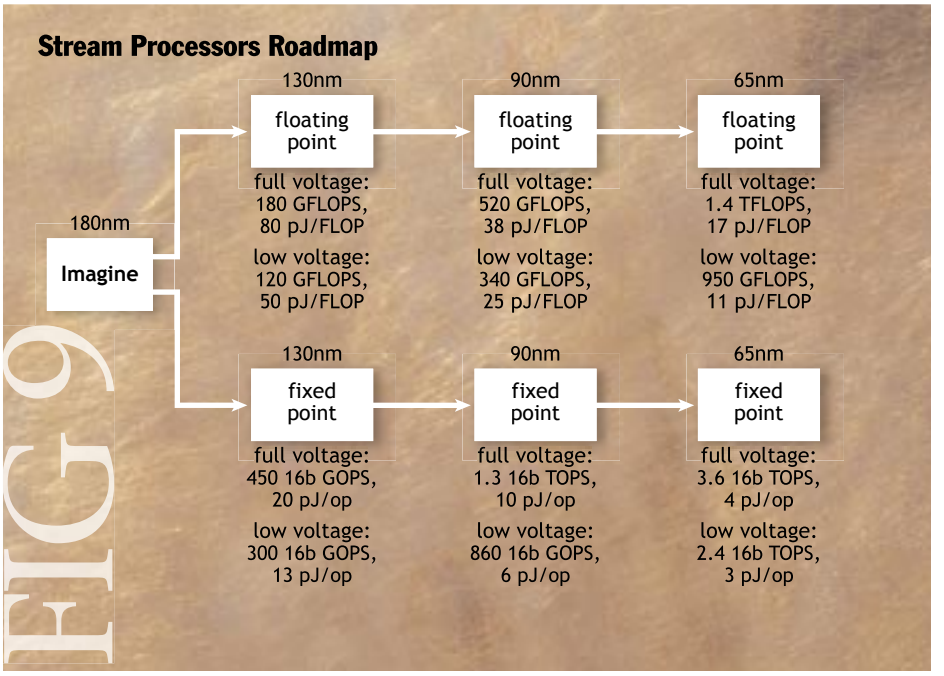


FIG 9

REFERENCES

1. Kanade, T., Yoshida, A., Oda, K., Kano, H., and Tanaka, M. A stereo machine for video-rate dense depth mapping and its new applications. *Proceedings of the 15th Computer Vision and Pattern Recognition Conference* (June 1996), 196–202.

2. Khailany B., Dally, W. J., Rixner, S., Kapasi, U. J., Owens, J. D., Towles, B. Exploring the VLSI scalability of stream processors. *Proceedings of the Ninth International Symposium on High Performance Computer Architecture* (Feb. 2003), 153–164.

Stream Processors

Programmability with Efficiency

3. Owens, J. D., Khailany, B., Towles, B., and Dally, W. J. Comparing Reyes and OpenGL on a stream architecture. *Siggraph/Eurographics Workshop on Graphics Hardware* (Sept. 2002), 47–56.
4. A high-end superscalar processor may consume 10 nJ or more per instruction because of much greater overheads required for acceleration techniques such as branch prediction, register renaming, and out-of-order execution.
5. Rixner, S., Dally, W. J., Khailany, B., Mattson, P., Kapasi, U. J., and Owens, J. D. Register organization for media processing. *Proceedings of the Sixth International Symposium on High Performance Computer Architecture* (Jan. 2000), 375–387.
6. Mattson, P., Dally, W. J., Rixner, S., Kapasi, U. J., and Owens, J. D. Communication scheduling. *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems* (Nov. 2000), 82–92.
7. Kapasi, U. J., Dally, W. J., Rixner, S., Mattson, P. R., Owens, J. D., and Khailany, B. Efficient conditional operations for data-parallel architectures. *Proceedings of the 33rd Annual IEEE/ACM International Symposium on Microarchitecture* (Dec. 2000), 159–170.
8. Khailany, B., Dally, W. J., Rixner, S., Kapasi, U. J., Mattson, P., Namkoong, J., Owens, J. D., Towles, B., and Chang, A. Imagine: Media processing with streams. *IEEE Micro 21*, 2 (Mar./Apr. 2001), 35–46.
9. See reference 2.
10. Robles, R. The cost/benefit ratio of ASICs in wireless baseband modems. Communications Design Conference (Oct. 2003); http://www.commdesignconference.com/archive/papers/2003/P212.htm.

LOVE IT, HATE IT? LET US KNOW

feedback@acmqueue.com or www.acmqueue.com/forums

WILLIAM J. DALLY is the Willard R. and Inez Kerr Bell Professor of Engineering at Stanford University. Dally has done pioneering development work at Bell Telephone Laboratories, Caltech, and the Massachusetts Institute of Technology, where he was a professor of electrical engineering and computer science. At Stanford University, his group has developed the Imagine processor, which introduced the concepts of stream processing and partitioned register organizations. Dally has worked with Cray Research and Intel to incorporate many of these innovations in commercial parallel computers, with Avici Systems to incorporate this technology into Internet routers, and he cofounded Velio Communications to commercialize high-speed signaling technology, and Stream Processors to commercialize stream processor technology. He is a fellow of IEEE and ACM, and has received numerous honors including the ACM Maurice Wilkes award. He has published more than 150 papers in these areas and is an author of the textbooks *Digital Systems Engineering* (Cambridge University Press, 1998) and *Principles and Practices of Interconnection Networks* (Morgan Kaufmann, 2003).

UJVAL J. KAPASI was expected to receive his doctorate from Stanford University in February 2004. His research interests include computer architecture, scientific computing, and language and compiler design. While at Stanford, he was an architect of the Imagine stream processor and contributed to the VLSI implementation of an Imagine prototype. Recently, he cofounded Stream Processors, which is commercializing the stream-processing technology developed at Stanford.

BRUCEK KHAILANY received his Ph.D. from Stanford University in 2002, where he was the principal VLSI designer of the Imagine stream processor. Recently, as a cofounder of Stream Processors, he has been working on the commercialization of stream processors in a variety of application areas. He is a member of IEEE and ACM, was an Intel Foundation fellowship recipient at Stanford, and received a BSEE from the University of Michigan in 1997.

JUNG HO AHN is a Ph.D. candidate in electrical engineering at Stanford University. His research interests include computer architecture, stream architecture, advanced memory design, and compiler design. Ahn received an MS in electrical engineering from Stanford. He is a student member of ACM and IEEE.

ABHISHEK DAS is a Ph.D. candidate in electrical engineering at Stanford University. He received his B.Tech in computer science and engineering from IIT Kharagpur, India. His research interests include compiler and language design, computer systems software, and computer architecture. He has worked on making TCP/IP feasible for the Bluetooth technology at the IBM Research Center, India. He is currently involved in the development and enhancement of the compiler and architecture for the Merrimac and Imagine architectures.