# Programmable Stream Processors


**Stream processing promises to bridge the gap between inflexible special-purpose solutions and current programmable architectures that cannot meet the computational demands of media-processing applications.**

*Ujval J. Kapasi*
Stanford University

*Scott Rixner*
Rice University

*William J. Dally*
*Brucek Khailany*
*Jung Ho Ahn*
Stanford University

*Peter Mattson*
Reservoir Labs

*John D. Owens*
University of California, Davis

The complexity of modern media processing, including 3D graphics, image compression, and signal processing, requires tens to hundreds of billions of computations per second. To achieve these computation rates, current media processors use special-purpose architectures tailored to one specific application. Such processors require significant design effort and are thus difficult to change as media-processing applications and algorithms evolve.

The demand for flexibility in media processing motivates the use of programmable processors. However, very large-scale integration constraints limit the performance of traditional programmable architectures. In modern VLSI technology, computation is relatively cheap—thousands of arithmetic logic units that operate at multigigahertz rates can fit on a modestly sized 1-cm$^2$ die. The problem is that delivering instructions and data to those ALUs is prohibitively expensive. For example, only 6.5 percent of the Itanium 2 die is devoted to the 12 integer and two floating-point ALUs and their register files[1]; communication, control, and storage overhead consume the remaining die area. In contrast, the more efficient communication and control structures of a special-purpose graphics chip, such as the Nvidia GeForce4, enable the use of many hundreds of floating-point and integer ALUs to render 3D images.

## STREAM PROCESSING

In part, such special-purpose media processors are successful because media applications have abundant parallelism—enabling thousands of computations to occur in parallel—and require minimal global communication and storage—enabling data to pass directly from one ALU to the next. A *stream* architecture exploits this locality and concurrency by partitioning the communication and storage structures to support many ALUs efficiently:

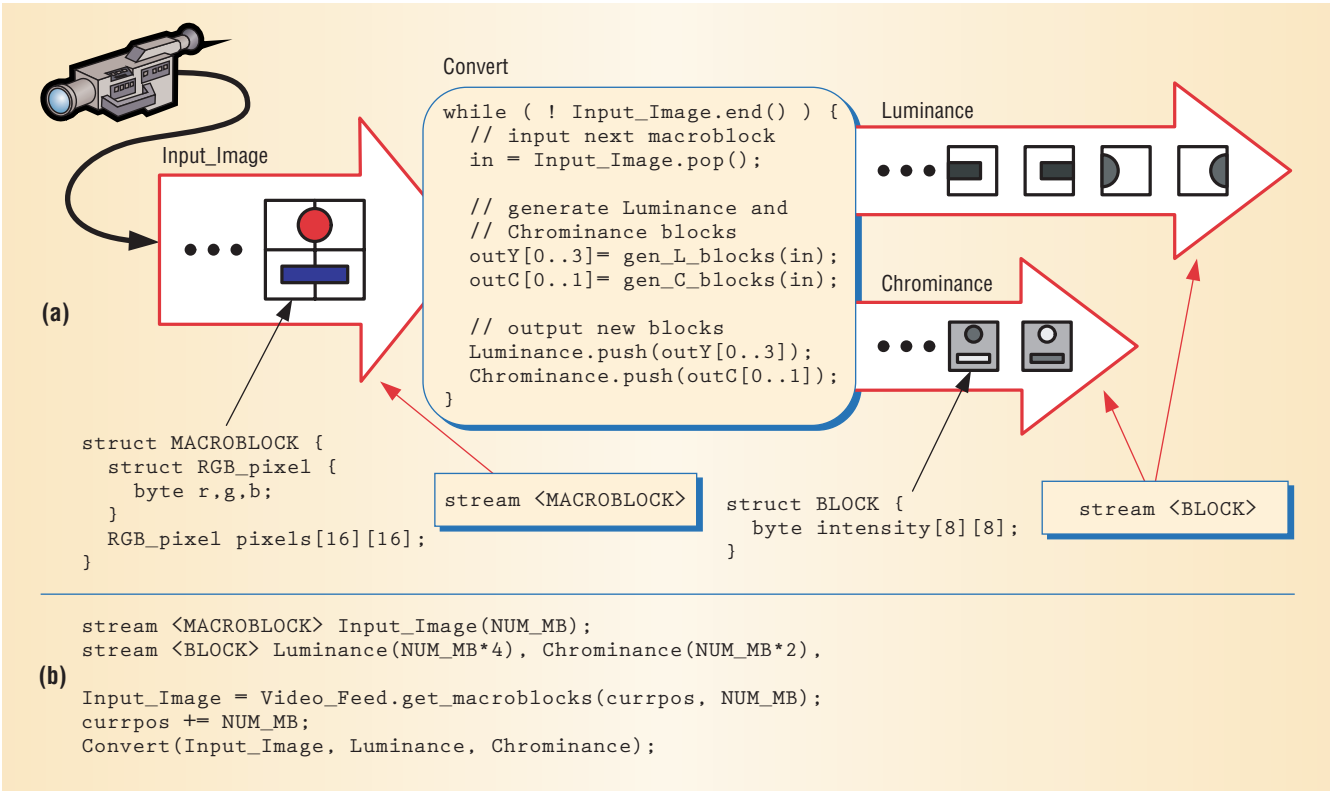
- operands for arithmetic operations reside in *local register files* (LRFs) near the ALUs, in much the same way that special-purpose architectures store and communicate data locally;
- streams of data capture coarse-grained locality and are stored in a *stream register file* (SRF), which can efficiently transfer data to and from the LRFs between major computations; and
- global data is stored off-chip only when necessary.

These three explicit levels of storage form a data *bandwidth hierarchy* with the LRFs providing an order of magnitude more bandwidth than the SRF and the SRF providing an order of magnitude more bandwidth than off-chip storage.

This bandwidth hierarchy is well matched to the characteristics of modern VLSI technology, as each level provides successively more storage and less bandwidth. By exploiting the locality inherent in media-processing applications, this hierarchy stores the data at the appropriate level, enabling hundreds of ALUs to operate at close to their peak rate.

Moreover, a stream architecture can support such a large number of ALUs in an area- and power-efficient manner. Modern high-performance micro-


Published by the IEEE Computer Society
0018-9162/03/$17.00 © 2003 IEEE




processors and digital signal processors continue to rely on global storage and communication structures to deliver data to the ALUs; these structures use more area and consume more power per ALU than a stream processor.

## STREAMS AND KERNELS

The central idea behind stream processing is to organize an application into *streams* and *kernels* to expose the inherent locality and concurrency in media-processing applications. In most cases, not only do streams and kernels expose desirable properties of media applications, but they are also a natural way of expressing the application. This leads to an intuitive programming model that maps directly to stream architectures with tens to hundreds of ALUs.

### Example application

Figure 1 illustrates input and output streams and a kernel taken from an MPEG-2 video encoder. Figure 1a shows how a kernel operates on streams graphically, while Figure 1b shows this process in a simplified form of StreamC, a stream programming language.

Input_Image is a stream that consists of image data from a camera. Elements of Input_Image are $16 \times 16$ pixel regions, or macroblocks, on which the Convert kernel operates. The kernel applies the same computation to the macroblocks in Input_Image, decomposing each one into six $8 \times 8$ blocks—four luminance blocks and two 4:1 subsampled chrominance blocks—and appends them to the Luminance and Chrominance output streams, respectively.

**Streams.** As Figure 1a shows, streams contain a set of elements of the same type. Stream elements can be simple, such as a single number, or complex, such as the coordinates of a triangle in 3D space. Streams need not be the same length—for example, the Luminance stream has four times as many elements as the input stream. Further, Input_Image could contain all of the macroblocks in an entire video frame, only a row of macroblocks from the frame, or even a subset of a single row. In the stream code in Figure 1b, the value of NUM_MB controls the length of the input stream.

**Kernels.** The Convert kernel consists of a loop that processes each element from the input stream. The body of the loop first pops an element from its input stream, performs some computation on that element, and then pushes the results onto the two output streams.

Kernels can have one or more input and output streams and perform complex calculations ranging from a few to thousands of operations per input element—one Convert implementation requires 6,464 operations per input macroblock to produce the six output blocks. The only external data that a kernel can access are its input and output streams. For example, Convert cannot directly access the data from the video feed; instead, the data must first be organized into a stream.

**Full application.** A full application, such as the MPEG-2 encoder, is composed of multiple streams

***Figure 1. Streams and a kernel from an MPEG-2 video encoder. (a) The Convert kernel translates a stream of macroblocks containing RGB pixels into streams of blocks containing luminance and chrominance pixels. (b) A StreamC program expresses the flow of streams through kernels textually.***

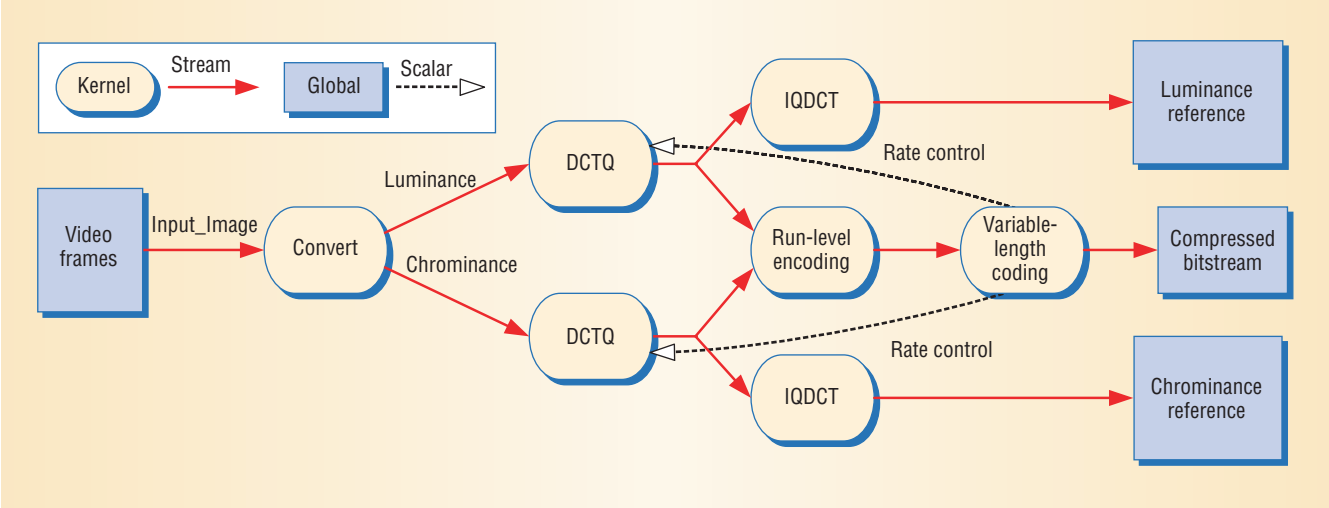



***Figure 2. MPEG-2 I-frame encoder mapped to streams and kernels. The encoder receives a stream of macroblocks from a frame in the video feed as input, and the first kernel (Convert) processes this stream. The discrete cosine transform (DCT) kernels then operate on the output streams produced by Convert. Q = quantization; IQ = inverse quantization.***

and kernels. This application compresses a sequence of video images into a single bitstream consisting of three types of frames: intracoded, predicted, and bidirectional. The encoder compresses I-frames using only information contained in the current frame, and it compresses P- and B-frames using information from the current frame as well as additional reference frames.

For example, Figure 2 shows one possible mapping of the portion of the MPEG-2 encoder application that encodes only I-frames into the stream-processing model. Solid arrows represent data streams, and ovals represent computation kernels.

The encoder receives a stream of macroblocks from the video feed as input (Input_Image), and the first kernel (Convert) processes this input. The discrete cosine transform and quantization (DCTQ) kernels then operate on the output streams produced by Convert. Upon execution of all the computation kernels, the application either transmits the compressed bitstream over a network or saves it for later use. The application uses the two reference streams to compress future frames. The representation of this entire application in a language such as StreamC is a straightforward extension of the code shown in Figure 1b.

## Locality and concurrency

By making communication between computation kernels explicit, the stream-processing model exposes both the locality and concurrency inherent in media-processing applications.

The model exposes locality by organizing communication into three distinct levels:

- *Local*. Temporary results that only need to be transferred between scalar operations within a kernel use local communication mechanisms. For example, temporary values in gen_L_blocks are only referenced within Convert. This type of communication cannot be used to transfer data between two different kernels.
- *Stream*. Data are communicated between computation kernels explicitly as data streams. In the MPEG-2 encoder, for example, the Luminance and Chrominance streams use this type of communication.
- *Global*. This level of communication is only for truly global data. This is necessary for communicating data to and from I/O devices, as well as for data that must persist throughout the application. For example, the MPEG-2 I-frame encoder uses this level of communication for the original input data from a video feed and for the reference frames, which must persist throughout the processing of multiple video frames in the off-chip dynamic RAM (DRAM).

By requiring programmers to explicitly use the appropriate type of communication for each data element, the stream model expresses the application's inherent locality. For example, the model only uses streams to transfer data between kernels and does not burden them with temporary values generated within kernels. Likewise, the model does not use global communication for temporary streams.

The stream model also exposes concurrency in media-processing applications at multiple levels:

- *Instruction-level parallelism (ILP)*. As in traditional processing models, the stream model can exploit parallelism between the scalar operations in a kernel function. For example, the operations in gen_L_blocks and gen_C_blocks can occur in parallel.
- *Data parallelism*. Because kernels apply the same computation to each element of an input stream, the stream model can exploit data parallelism by operating on several stream elements at the same time. For example, the model parallelizes the main loop in Convert so that multiple computation elements can each decompose a different macroblock.

- *Task parallelism*. Multiple computation tasks, including kernel execution and stream data transfers, can execute concurrently as long as they obey dependencies in the stream graph. For example, in the MPEG-2 I-frame application, the two DCTQ kernels could run in parallel.

Languages such as StreamC and StreamIt[2] that implement the stream model let programmers express communication explicitly at each level. A compiler can easily extract and optimize the locality and concurrency, as it does not need to discover them on its own.[3]

## STREAM INSTRUCTION-SET ARCHITECTURE

In a conventional scalar instruction set, communication is implicit, making it difficult to exploit locality, and computation is expressed sequentially, making it difficult to exploit concurrency. In contrast, a stream instruction set explicitly communicates to the hardware the locality and concurrency that the stream model exposes.

Figure 3 shows the architecture of a baseline programmable stream processor, which consists of an application processor, a stream register file, and stream clients. The SRF serves as a communication link by streaming data between clients, as long as the data does not exceed its storage capacity. The two stream clients in the baseline stream processor—a programmable kernel execution unit and an off-chip DRAM interface—either consume data streams from the SRF or produce data streams to the SRF. The KEU executes kernels and provides local communication for operations within the kernels, while the off-chip DRAM interface provides access to global data storage.

The architecture can also support other stream clients, including interfaces to I/O devices, an interface to an off-chip interconnection network, or specialized kernel units such as a variable-length Huffman encoder. Multiple instances of any particular stream client are also possible.

### Application processor ISA

The application processor executes application code like that in Figure 1b. The application processor's RISC-like instruction set is augmented with stream-level instructions to control the flow of data streams through the system. The application processor sequences these instructions and issues them to the stream clients, including the DRAM interface and the KEU.

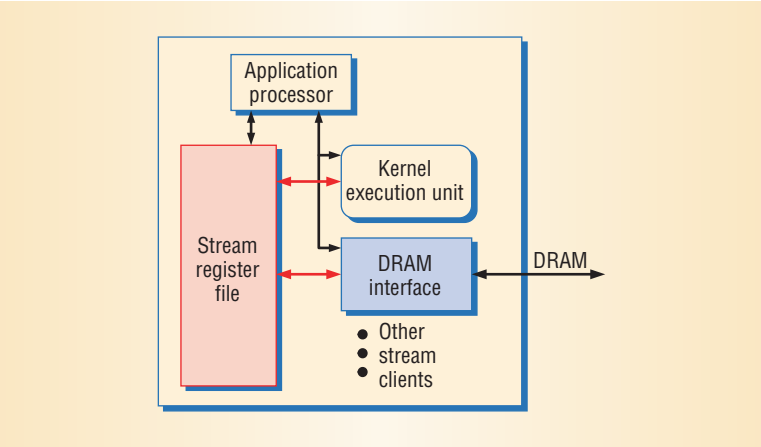



***Figure 3. Baseline programmable stream processor. The SRF streams data between clients—in this case the KEU, which is responsible for executing kernels and providing local communication for operations within the kernels, and the off-chip DRAM interface, which provides access to storage for global data.***

The DRAM interface supports two stream-level instructions—load_stream and store_stream—that transfer an entire stream between off-chip DRAM and the SRF. Additional DRAM interface arguments can specify noncontiguous access patterns, such as nonunit strides and indirect access. The DRAM interface client also supports optional caches to increase access bandwidth.

The programmable KEU supports two stream-level instructions as well. The load_kernel instruction loads a compiled kernel function into local instruction storage within the KEU. This typically occurs only the first time a kernel is executed; on subsequent kernel invocations, the code is already available in the KEU. The run_kernel instruction causes the KEU to start executing instructions that are encoded in its own instruction-set architecture, which is distinct from the application processor ISA.

### Kernel execution unit ISA

Instructions in the KEU ISA control the functional units and register storage within the KEU, similar to typical RISC instructions. However, unlike a RISC ISA,

- KEU instructions do not have access to arbitrary memory locations, as all external data must be read from or written to streams; and
- special communication instructions explicitly handle data dependencies between the computations of different output elements.

The first constraint preserves locality, while the communication mechanisms make it easier to exploit concurrency in applications that would otherwise need to reorganize data through the memory system. Such an ISA maximizes stream architecture performance.

### Stream ISA

A stream processor's ISA encapsulates both the application processor ISA and the KEU ISA. Accordingly, a compiler translates high-level stream languages directly to the stream processor's ISA.

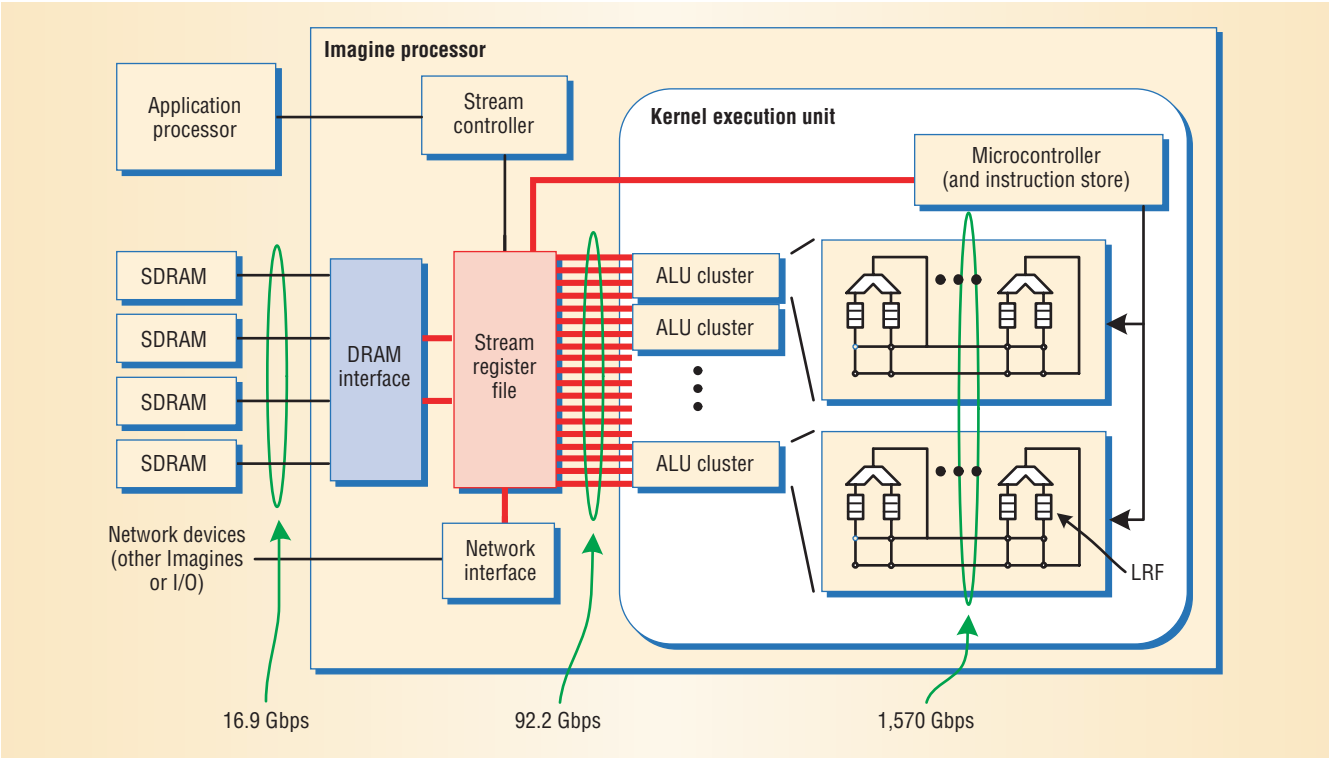



***Figure 4. Imagine stream processor. The on-chip bandwidths correspond to an operating frequency of 180 MHz.***

For example, it compiles the Convert kernel code shown in Figure 1a to the KEU ISA offline. The compiler then compiles the application-level code shown in Figure 1b to the application processor ISA. The application processor executes this code and moves the precompiled kernel code to the KEU instruction memory during execution.

Assuming that the portion of the video feed the processor is encoding currently resides in external DRAM, the three lines of application code in Figure 1b would result in the following sequence of operations (ii = Input_Image, lum = Luminance, and chrom = Chrominance):

```
load_stream    ii_SRF_loc,
               video_DRAM_loc, LEN

add            video_DRAM_loc,
               video_DRAM_loc, LEN

run_kernel     convert_KEU_loc,
ii_SRF_loc, LEN,
               lum_SRF_loc,
               chrom_SRF_loc
```

The first instruction loads a portion of the raw video feed into the SRF. The second instruction is a regular scalar operation that only updates local state in the application processor. The final instruction causes the KEU to start executing the operations for the Convert kernel. Similar instructions are required to complete the rest of the MPEG-2 I-frame application pipeline.

Interestingly, a traditional compiler can only perform manipulations on simple scalar operations that tend to benefit local performance only. A stream compiler, on the other hand, can manipulate instructions that operate on entire streams, potentially leading to greater performance benefits.

To be a good compiler target for high-level stream languages, the stream processor's ISA must express the original program's locality and concurrency. To this end, three distinct address spaces in the stream ISA support the three types of communication that the stream model uses. The ISA maps local communication within kernels to the address space of the local registers within the KEU, stream communication to the SRF address space, and global communication to the off-chip DRAM address space.

To preserve the stream model's locality and concurrency, the processor's ISA also exploits the three types of parallelism. To exploit ILP, the processor provides multiple functional units within the KEU. Because a kernel applies the same function to all stream elements, the compiler can exploit data parallelism using loop unrolling and software pipelining on kernel code. The KEU can exploit additional data parallelism using parallel hardware. To exploit task parallelism, the processor supports multiple stream clients connected to the SRF.

## PUTTING IT ALL TOGETHER: IMAGINE

*Imagine* (http://cva.stanford.edu/imagine/), designed and prototyped at Stanford University, is the first programmable streaming-media processor that implements a stream instruction set.[4] At a controlled voltage and temperature, the chips operate at 288 MHz, at which speed the peak performance is 11.8

billion 32-bit floating-point operations per second, or 23.0 billion 16-bit operations per second. However, in its current experimental setup the processor is seated in a socket to aid debugging, which limits performance to 180 MHz. At the most energy-efficient operating voltage, 1.2 V, the prototype achieves 4.7 billion 16-bit operations per joule.

The architecture, shown in Figure 4, contains all the basic stream hardware of the baseline processor, including an application processor, an SRF, and stream clients. The stream clients include a KEU and DRAM interface, as well as a network interface. The KEU consists of eight arithmetic clusters that are controlled by a single microcontroller in a single-instruction, multiple-data (SIMD) fashion.

Imagine supports 48 ALUs on a 2.6-cm$^2$ die in a 1.5-V, 0.15-µm complementary metal-oxide semiconductor Texas Instruments process using a full standard-cell design flow. The 32-bit ALUs can perform both integer and floating-point operations. Figure 5 shows the layout of the 21-million-transistor die. The ALU clusters consume 26 percent of the die, the SRF consumes 16 percent, and the memory interface, the network interface, and control and routing consume the rest of the chip area.

To simplify the design and implementation of the Imagine architecture, the prototype system splits the functionality of the application processor into two portions. The on-chip *stream controller* (SC) only handles the sequencing of operations to the stream clients. An external processor executes the application-level code and hands off any operations destined for stream clients to the stream controller through the *host interface* (HI).

## Bandwidth hierarchy

The stream model exploits locality and concurrency while satisfying VLSI technology constraints. In modern VLSI, global communication is far more expensive than local communication. For example, the maximum data bandwidth provided by a global register file is an order of magnitude higher than the bandwidth provided by off-chip memory. Further, the total bandwidth provided by tens to hundreds of local memories, each servicing a few nearby ALUs, is another one to two orders of magnitude greater.

Accordingly, as Figure 4 shows, Imagine provides three levels of storage that form a data bandwidth hierarchy. The top tier consists of 9.5 Kbytes in a set of distributed local register files (LRFs) in the KEU (1,570 Gbps), followed by the 128-Kbyte SRF (92.2 Gbps) and the off-chip DRAM interface (16.9 Gbps). The stream ISA maps the three types of communication in the stream model—local, stream, and global—to these three tiers. The locality in media applications ensures that most of the data bandwidth demand will be for the LRFs, much less will be for the SRF, and only the accesses to persistent data structures will access off-chip DRAM. Thus, the bandwidth hierarchy on Imagine is well suited to exploit the application locality captured by the stream model.

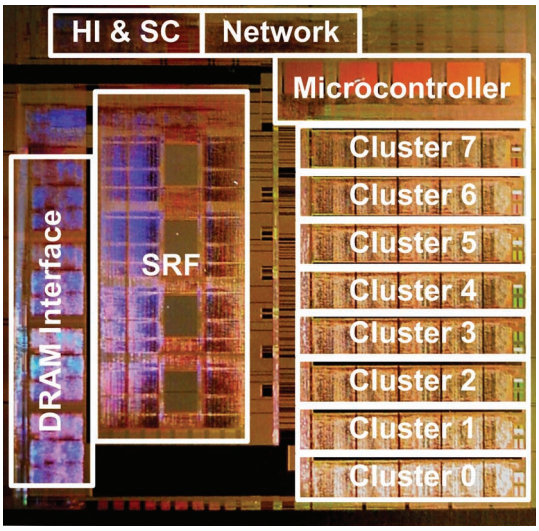



***Figure 5. Imagine die layout. The ALU clusters consume 26 percent of the die area, the SRF consumes 16 percent, and the memory interface, the network interface, and control and routing consume the rest. HI = host interface; SC = stream controller.***

The bandwidths achieved for the three levels of the hierarchy for the MPEG-2 encoder demonstrate the amount of locality Imagine exploits: 592 Gbps via the LRFs, 6.47 Gbps via the SRF, and 1.26 Gbps via DRAM. This ratio of 470:5:1 means that 98.7 percent of all data accesses are captured locally in the LRFs, and only 0.21 percent of the accesses use off-chip DRAM.

Further, Imagine does not require any extra transistors or waste any energy to dynamically capture this locality, such as with a cache or bypass network. The original stream program expressed the locality and the ISA encoded it.

Note that these results are for the full MPEG-2 encoding application, which includes encoding both I- and P-frames, but without variable-length Huffman encoding. Huffman encoding's sequential nature makes it better suited for the application processor.

## SIMD clusters

Imagine also successfully exploits the concurrency of stream programs at several levels. As Figure 4 shows, a single microcontroller controls multiple clusters in a SIMD fashion. The SIMD clusters are particularly effective at capturing data parallelism. For example, Imagine executes the Convert kernel by processing a different macroblock in each cluster. Again, the processor does not require extra hardware or energy to discover this parallelism, as it is present in the stream program. By structuring the original computation as a kernel, the program-

**Table 1. Performance and power dissipation of the Imagine stream processor.**

| Application | Operation types | 180 MHz, 2.0 V | | 96 MHz, 1.2 V | |
|---|---|---|---|---|---|
| | | Performance (giga operations per second) | Core power (watts) | Performance (giga operations per second) | Core power (watts) |
| MPEG-2 encode | 32-, 16-, and 8-bit integer | 6.2 | 8.0 | 3.5 | 1.6 |
| QR decomposition | 32-bit floating point | 4.2 | 9.8 | 2.2 | 1.9 |
| Coherent side-lobe cancellation | 32-bit floating point | 2.8 | 10.0 | 1.4 | 1.9 |

mer makes it explicit that the processor can produce each output element in parallel.

A good measure of the amount of data parallelism that Imagine can successfully exploit is the speedup of an application running on an eight-cluster Imagine compared to an identical stream processor with a single cluster. To take an example from the MPEG-2 application, an $8 \times 8$ DCT kernel runs 7.7 times faster. Imagine can also exploit data parallelism via 8- and 16-bit parallel subword operations.

To exploit instruction-level parallelism, each cluster contains six ALUs controlled by a compiled very long instruction word (VLIW) kernel program. The processor's ILP can be measured by the number of instructions executed every cycle within each cluster. The MPEG-2 application achieves 4.4 operations per cycle per cluster.

Finally, Imagine exploits task parallelism on-chip because it can transfer two streams via the DRAM interface and execute a kernel concurrently. Compared to a stream processor that does not support this concurrency, Imagine executes the MPEG-2 application 1.2 times faster and a coherent side-lobe cancellation application 1.7 times faster.

## Processor performance

Table 1 demonstrates the performance of the Imagine stream processor on three typical media applications. Performance was measured at an operating frequency of 180 MHz, and at a more power-efficient operating point of 1.2 V at 96 MHz. At 180 MHz, Imagine achieves between 2.8 and 6.2 billion operations per second on actual applications. For floating-point applications, this is approximately 40 to 50 percent of the peak performance of 7.4 Gflops. At the more efficient operating point, the processor achieves a power efficiency between 0.7 and 2.2 giga operations per joule.

Imagine achieves a maximum power efficiency of 2.4 Gflops per watt, measured on an application exercising peak performance, which compares favorably to other programmable floating-point processors. General-purpose microprocessors rely on deep pipelining for high clock frequencies and speculative execution to achieve high performance rates, resulting in low power efficiencies. For example, at 3.08 GHz, the Pentium 4 achieves a peak performance of 12 Gflops at 80 watts,[5] more than an order of magnitude worse than Imagine when normalized to the same technology.

Digital signal processors (DSPs), on the other hand, have kept clock rates and pipeline lengths small for power efficiency, but this limits their peak performance. For example, the 225-MHz Texas Instruments C67x provides a peak of 1.35 Gflops[6] and, when normalized to the same process technology, is two to three times less power efficient than Imagine. Moreover, using more aggressive circuit designs and power-saving techniques would significantly improve Imagine's power efficiency.

Finally, not only does a stream architecture efficiently support 48 ALUs in current VLSI technology, it will also be able to effectively scale to much higher levels of performance in the near future. This is especially true compared to DSP and even vector architectures, as the "Stream versus Vector Processors" sidebar describes. A recent study shows that programmable stream processors with a microarchitecture similar to Imagine will be able to support more than 1,000 ALUs by 2007.[7] Such a processor would have a peak performance of more than 1 trillion operations per second while dissipating less than 10 watts.

The Imagine media processor validates the hypothesis that careful management of bandwidth and parallelism, from the programming language to the hardware, results in both high performance and high performance per unit of power. We envision that the future study of stream processors will follow three major directions: increasing the number of application domains that benefit from stream architectures' high performance and power efficiency, automatic mapping of the stream model to multinode systems, and improving high-level stream languages and compilers.

Research into advanced compiler technology for stream languages and architectures is being conducted as part of Reservoir Labs' R-Stream compiler project (www.reservoir.com/r-stream.php). Further research that extends Imagine's processing capabilities is also under way as part of Stanford University's Merrimac project (http://merrimac.stanford.edu/). The goals of this *streaming supercomputer*[8] are to demonstrate that stream processing is well suited for

## Stream versus Vector Processors

Stream processors share with vector processors the ability to hide latency, amortize instruction overhead, and expose data parallelism by operating on large aggregates of data. Vector processors such as the Cray family (the Cray 1[1] through the X-1) and the recent VIRAM1,[2] an implementation of the Berkeley Intelligent RAM project, use vector instructions to load, store, and operate on a fixed-length vector. The length of each vector is *VL* data words.

Loading a vector with a single instruction hides memory latency with parallelism by loading the vector's remaining words while waiting for the first to return from memory. Amortizing the costs of instruction fetch, issue, and decode over *VL* operations reduces overhead. Having each instruction specify *VL* operations, some of which may be performed in parallel on multiple vector pipes, exposes data parallelism.

In a similar manner, a stream processor such as Imagine hides memory latency by fetching a stream of records with a single stream load instruction. It also executes a kernel on entire streams of records, amortizing the overhead of the run_kernel instruction and exposing data parallelism that can be exploited by operating on records in parallel using multiple processing clusters.

Stream processors extend vector processors' capabilities in two ways. First, they add a layer to the register hierarchy by splitting the functions of a vector processor's vector register file between the local register files and the stream register file. The LRFs require only a modest amount of capacity, allowing them to be built with high aggregate bandwidth to support a large number of arithmetic logic units. Because the SRF is relieved of the task of forwarding data to and from the ALUs, its bandwidth is an order of magnitude less than the LRFs, making it economical to build SRFs large enough to exploit coarse-grained locality.

Second, a stream-processing kernel performs all of the operations for one record of the input before moving on to the next record. This reduces the number of intermediate variables stored at any single moment in the LRFs by a factor of *VL*—strictly speaking, *VL* divided by the number of clusters.

A stream processor starts with the latency hiding and instruction efficiency of a vector processor. By adding a level of register hierarchy and a level of instruction sequencing, it can reduce bandwidth demands on the memory system by an order of magnitude for many applications and thus can support an order of magnitude more ALUs than a vector processor with the same memory bandwidth.

### References


1. R.M. Russell, "The Cray-1 Computer System," *Comm. ACM,* vol. 21, no. 1, 1978, pp. 63-72.
2. C. Kozyrakis, *A Media-Enhanced Vector Architecture for Embedded Memory Systems,* tech. report UCB-CSD-99-1059, Computer Science Division, Univ. of California, Berkeley, 1999.


scientific computing applications, that stream-processing hardware and software systems can effectively scale to 16,384 nodes, and that a streaming architecture substrate for large-scale scientific machines will provide more than an order of magnitude improvement in performance per cost over what today's symmetric multiprocessor clusters deliver. This new research promises to improve the state of the art in high-end scientific computing, as the Imagine processor has done in the media-processing domain. ■

### References


1. S.D. Naffziger et al., "The Implementation of the Itanium 2 Microprocessor," *IEEE J. Solid-State Circuits,* Nov. 2002, pp. 1448-1460.
2. W. Thies, M. Karczmarek, and S. Amarasinghe, "StreamIt: A Language for Streaming Applications," *Proc. 11th Int'l Conf. Compiler Construction,* LNCS 2304, Springer-Verlag, 2002, pp. 179-196.
3. U.J. Kapasi et al., *Stream Scheduling,* Concurrent VLSI Architecture Tech. Report 122, Computer Systems Laboratory, Stanford Univ., Stanford, Calif., 2002.
4. S. Rixner et al., "A Bandwidth-Efficient Architecture for Media Processing," *Proc. 31st Ann. ACM/IEEE Int'l Symp. Microarchitecture,* IEEE CS Press, 1998, pp. 3-13.
5. D. Sager et al., "A 0.18µm CMOS IA32 Microprocessor with a 4-GHz Integer Execution Unit," *2001 IEEE Int'l Solid-State Circuits Conf. Digest of Technical Papers,* IEEE Standards Office, 2001, pp. 324-325.
6. Texas Instruments, *TMS320C6713 Floating-Point Digital Signal Processor,* Datasheet SPRS186D, Dec. 2001, rev. May 2003; http://focus.ti.com/lit/ds/symlink/tms320c6713.pdf.
7. B. Khailany et al., "Exploring the VLSI Scalability of Stream Processors," *Proc. 9th Int'l Symp. High-Performance Computer Architecture,* IEEE CS Press, 2003, pp. 153-164.
8. W.J. Dally, P. Hanrahan, and R. Fedkiw, *A Streaming Supercomputer,* white paper, Computer Systems Laboratory, Stanford Univ., Stanford, Calif., 2001.



***Ujval J. Kapasi*** *is a PhD candidate in electrical engineering at Stanford University. His research interests include computer architecture, scientific computing, and language and compiler design. Kapasi received an MS in electrical engineering from Stanford University. He is a member of the IEEE and the ACM. Contact him at ujk@cva.stanford.edu.*

***Scott Rixner*** *is an assistant professor of computer science and electrical and computer engineering at Rice University, where he is a co-leader of the Rice*

*Computer Architecture Group. His research interests include media, network, and communications processing; memory system architecture; and the interaction between operating systems and network server architectures. Rixner received a PhD in electrical engineering from the Massachusetts Institute of Technology. He is a member of the ACM. Contact him at rixner@rice.edu.*

***William J. Dally** is a professor of electrical engineering and computer science and a member of the Computer Systems Laboratory at Stanford University, where he heads the Concurrent VLSI Architecture Group. His research interests include network architecture, multicomputer architecture, media-processor architecture, and high-speed CMOS signaling. Dally received a PhD in computer science from the California Institute of Technology. He is a Fellow of the IEEE, a Fellow of the ACM, and a recipient of the ACM Maurice Wilkes Award. Contact him at dally@stanford.edu.*

***Brucek Khailany** is a postdoctoral scholar in electrical engineering at Stanford University. His research interests include power-efficient computer architecture, media processing, circuits, and computer arithmetic. Khailany received a PhD in electrical engineering from Stanford University. He is a member of the IEEE and the ACM. Contact him at khailany@cva.stanford.edu.*

***Jung Ho Ahn** is a PhD candidate in electrical engineering at Stanford University. His research interests include computer architecture, stream computing, compiler design, and advanced memory design. Ahn received an MS in electrical engineering from Stanford University. He is a student member of the IEEE. Contact him at gajh@cva.stanford.edu.*

***Peter Mattson** is a managing engineer at Reservoir Labs, an advanced technology services company with offices in New York City and Portland, Ore. His research interests include high-performance languages and compilers. Mattson received a PhD in electrical engineering from Stanford University. He is a member of the IEEE and the ACM. Contact him at mattson@reservoir.com.*

***John D. Owens** is an assistant professor of electrical and computer engineering at the University of California, Davis. His research interests include sensor networks, graphics and graphics architectures, stream computing, and computer architecture. Owens received a PhD in electrical engineering from Stanford University. Contact him at jowens@ece.ucdavis.edu.*