# IMAGINE: MEDIA PROCESSING WITH STREAMS

THE POWER-EFFICIENT IMAGINE STREAM PROCESSOR ACHIEVES PERFORMANCE DENSITIES COMPARABLE TO THOSE OF SPECIAL-PURPOSE EMBEDDED PROCESSORS. EXECUTING PROGRAMS MAPPED TO STREAMS AND KERNELS, A SINGLE IMAGINE PROCESSOR IS EXPECTED TO HAVE A PEAK PERFORMANCE OF 20 GFLOPS AND SUSTAIN 18.3 GOPS ON MPEG-2 ENCODING.

Brucek Khailany
William J. Dally
Ujval J. Kapasi
Peter Mattson
Jinyung Namkoong
John D. Owens
Brian Towles
Andrew Chang
Stanford University

Scott Rixner
Rice University

•••••• Media-processing applications, such as signal processing, 2D- and 3D-graphics rendering, and image and audio compression and decompression, are the dominant workloads in many systems today. The real-time constraints of media applications demand large amounts of absolute performance and high performance densities (performance per unit area and per unit power). Therefore, media-processing applications often use special-purpose (custom), fixed-function hardware. General-purpose solutions, such as programmable digital signal processors (DSPs), offer increased flexibility but achieve performance density levels two or three orders of magnitude worse than special-purpose systems.

One reason for this performance density gap is that conventional general-purpose architectures are poorly matched to the specific properties of media applications. These applications share three key characteristics. First, operations on one data element are largely independent of operations on other elements, resulting in a large amount of data parallelism and high latency tolerance. Second, there is little global data reuse. Finally, the applications are computationally intensive, often performing 100 to 200 arithmetic operations for each element read from off-chip memory.

Conventional general-purpose architectures don't efficiently exploit the available data parallelism in media applications. Their memory systems depend on caches optimized for reducing latency and data reuse. Finally, they don't scale to the numbers of arithmetic units or registers required to support a high ratio of computation to memory access. In contrast, special-purpose architectures take advantage of these characteristics because they effectively exploit data parallelism and computational intensity with a large number of arithmetic units. Also, special-purpose processors directly map the algorithm's dataflow graph into hardware rather than relying on memory systems to capture locality.

Another reason for the performance density gap is the constraints of modern technology. Modern VLSI computing systems are limited by communication bandwidth rather than arithmetic. For example, in a contemporary 0.15-micron CMOS technology, a 32-bit integer adder requires less than 0.05 $mm^2$ of chip area. Hundreds to thousands of these arithmetic units fit on an inexpensive 1-$cm^2$ chip. The challenge is supplying them with instructions and data. General-purpose processors that rely on global structures such as large multiported register files to provide

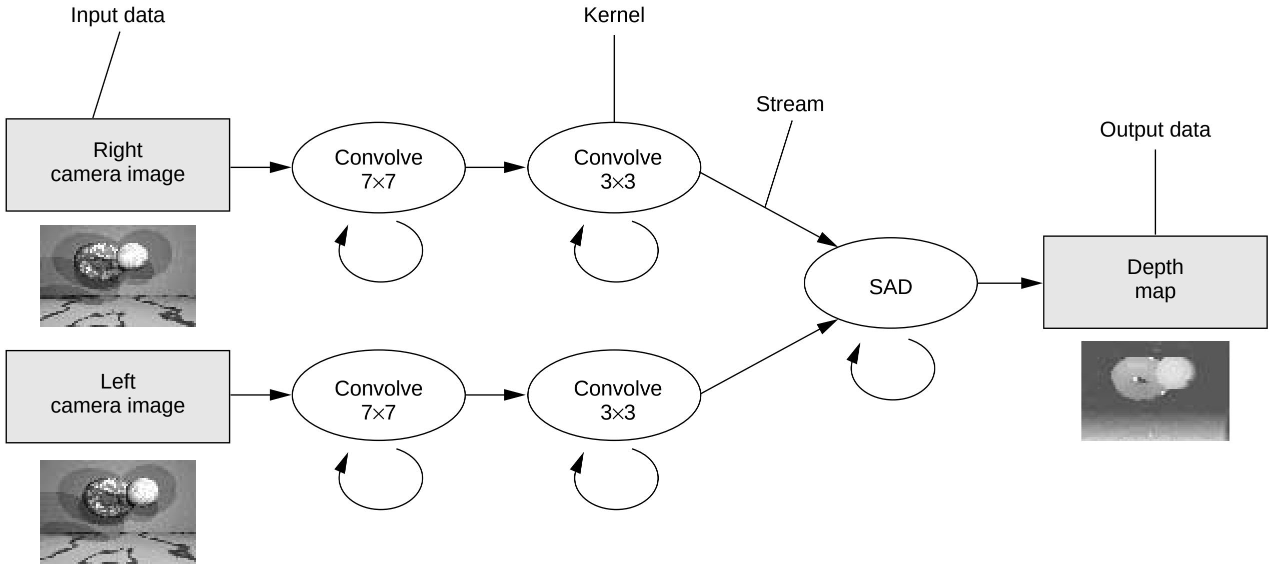


Figure 1. Stereo depth extraction.

data bandwidth cannot scale to the number of arithmetic logic units (ALUs) required for high performance rates in media applications. In contrast, special-purpose processors have many ALUs connected by dedicated wires and buffers to provide needed data and control bandwidth. However, special-purpose solutions lack the flexibility to work effectively on a wide application space.

To provide programmability yet maintain high performance densities, we developed the Imagine stream processor at Stanford University. Imagine consists of a programming model, software tools, and an architecture, all designed to operate directly on streams. The stream programming model exposes the locality and concurrency in media applications so they can be exploited by the stream architecture. This architecture uses a novel register file organization that supports 48 floating-point arithmetic units. We expect a prototype Imagine processor to achieve 18.3 giga operations per second (GOPS) in MPEG-2 encoding applications, corresponding to 105 frames per second on a 720 × 480-pixel, 24-bit color image while dissipating 2.2 W.

## Stream programming model

The stream programming model allows simple control, makes communication explicit, and exposes the inherent parallelism of media applications. A stream program organizes data as streams and expresses all computation as kernels. A stream is a sequence of similar elements. Each stream element is a record, such as 21-word triangles or 8-bit pixels. A kernel consumes a set of input streams, performs a computation, and produces a set of output streams. Streams passing among multiple computation kernels form a stream program.

Figure 1 shows how we map stereo depth extraction,[1] a typical image-processing application, to the stream programming model. Stereo depth extraction calculates the disparity between objects in a left and right grayscale camera image pair. It outputs a depth map corresponding to each object's distance from the cameras (in the example depth map, brighter pixels are closer to the camera). The input camera images are formatted as streams; in the example, a single stream contains a row of pixels from the image. The row streams flow though two convolution kernels, which produce a filtered stream. The filtered streams then pass through a sum-of-absolute-differences (SAD) kernel, which produces a stream containing a row of the final depth map. Each kernel also outputs a stream of partial sums (represented by the circular arrows in the diagram) needed by the same kernel as it processes future rows.

Stereo depth extraction possesses important characteristics common to all media-processing applications:

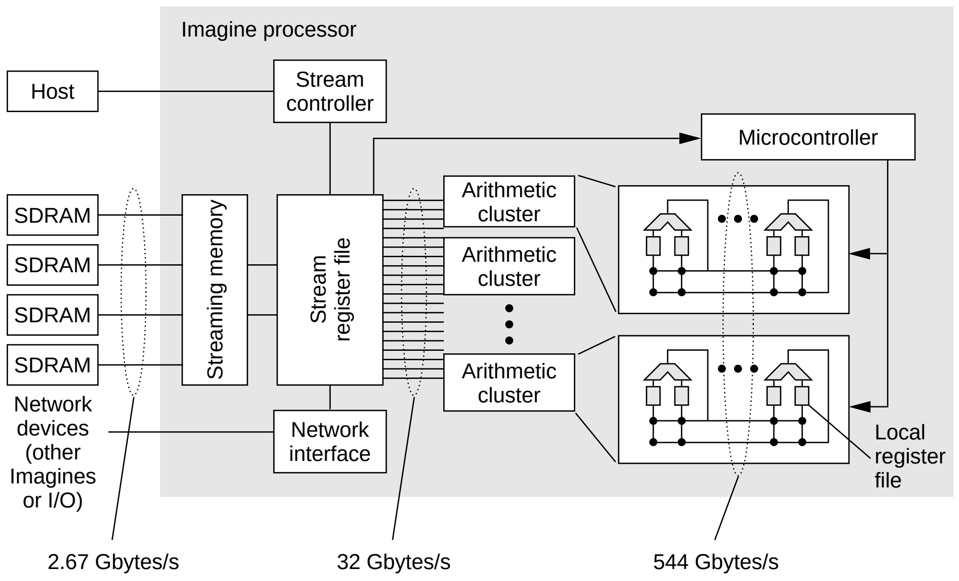


Figure 2. Imagine stream architecture.

- *Little data reuse.* Pixels are read once from memory and are not revisited.
- *High data parallelism.* The same set of operations independently computes all pixels in the output image.
- *Computationally intensive.* The application requires 60 arithmetic operations per memory reference.

Mapping an application to streams and kernels exposes these characteristics so that hardware can easily exploit them. Streams expose an application's inherent data parallelism. The flow of streams between kernels exposes the application's communication requirements.

## Stream architecture

Imagine is a programmable processor that directly executes applications mapped to streams and kernels. Figure 2 diagrams the Imagine stream architecture. The processor consists of a 128-Kbyte stream register file (SRF), 48 floating-point arithmetic units in eight arithmetic clusters controlled by a microcontroller, a network interface, a streaming memory system with four SDRAM channels, and a stream controller. Imagine is controlled by a host processor, which sends it stream instructions. The following are the main stream instructions:

- *Load* transfers streams from off-chip DRAM to the SRF.
- *Store* transfers streams from the SRF to off-chip DRAM.
- *Receive* transfers streams from the network to the SRF.
- *Send* transfers streams from the SRF to the network.
- *Cluster op* executes a kernel in the arithmetic clusters that reads inputs streams from the SRF, computes output streams, and writes the output streams to the SRF.
- *Load microcode* loads streams consisting of kernel microcode—576-bit very long instruction word (VLIW) instructions—from the SRF into the microcontroller instruction store (a total of 2,048 instructions).

Imagine supports streams up to 32K (32,768) words long. The host processor sends stream instructions to the stream controller. When instructions are ready for issue, the stream controller decodes them and issues the necessary commands to other on-chip modules.

Each arithmetic cluster, detailed on the right side of Figure 2, contains eight functional units. A small two-ported local register file (LRF) connects to each input of each functional unit. An intracluster switch connects the outputs of the functional units to the inputs of the LRFs. During kernel execution, the same VLIW instruction is broadcast to all eight clusters in single-instruction, multiple-data (SIMD) fashion.

Kernels typically loop through all input-stream elements, performing a compound stream operation on each element. A compound stream operation reads an element from its input stream(s) in the SRF and computes a series of arithmetic operations (for example, matrix multiplication or convolution). Then it appends the results to output streams that are transferred back to the SRF. A compound stream operation stores all temporary data in the LRFs. It sends only final results to the SRF.

Compound stream operations are compound in the sense that they perform multiple arithmetic operations on each stream element. In contrast, conventional vector operations perform a single arithmetic operation on each vector element and store the results back in the vector register file after each operation.

## Stream application example

The stereo depth extraction application presented in Figure 1 demonstrates how the Imagine stream architecture functions. The convolution stage performs a pair of 2D convolutions on both the left and right camera images, first using a $7 \times 7$ filter and then a $3 \times 3$ filter. (The application uses two separate filters to reduce the operation count.) Figure 3a shows the application-level pseudocode for the convolution of one image. Figure 3b shows how the commands in pseudocode lines 3 through 6 map to stream operations on the Imagine architecture (circled numbers indicate pseudocode lines).

The stream controller dynamically schedules each stream instruction. When the required resources are ready and interinstruction dependencies are satisfied, the stream controller issues the following operations:

*Load operation.* Line 3 translates into a stream Load instruction that causes a stream transfer from off-chip DRAM to the SRF. In this case, the stream being transferred contains a row of the image. Each element in the stream is a pixel from the source image, and the stream length corresponds to the image width. During the Load operation, the streaming memory system generates addresses corresponding to the pixel elements' locations, which are laid out sequentially in memory. The memory system then reorders these accesses to maximize DRAM bandwidth. As the Load operation executes, elements are written back to the SRF in their original order. At completion of the operation, the SRF contains a stream consisting of all the pixels in a row.

*Convolve7×7 kernel execution.* Next, the stream controller issues pseudocode line 4, causing the convolve7×7 kernel (Figure 3c) to execute. This kernel has two input streams: the pixel stream just loaded from memory (src) and a stream containing the partial sums produced earlier in the application (old_partials7). During kernel execution, the microcontroller fetches VLIW microcode instructions from the instruction store every cycle. Then the microcontroller sends these instructions to all eight arithmetic clusters in parallel.

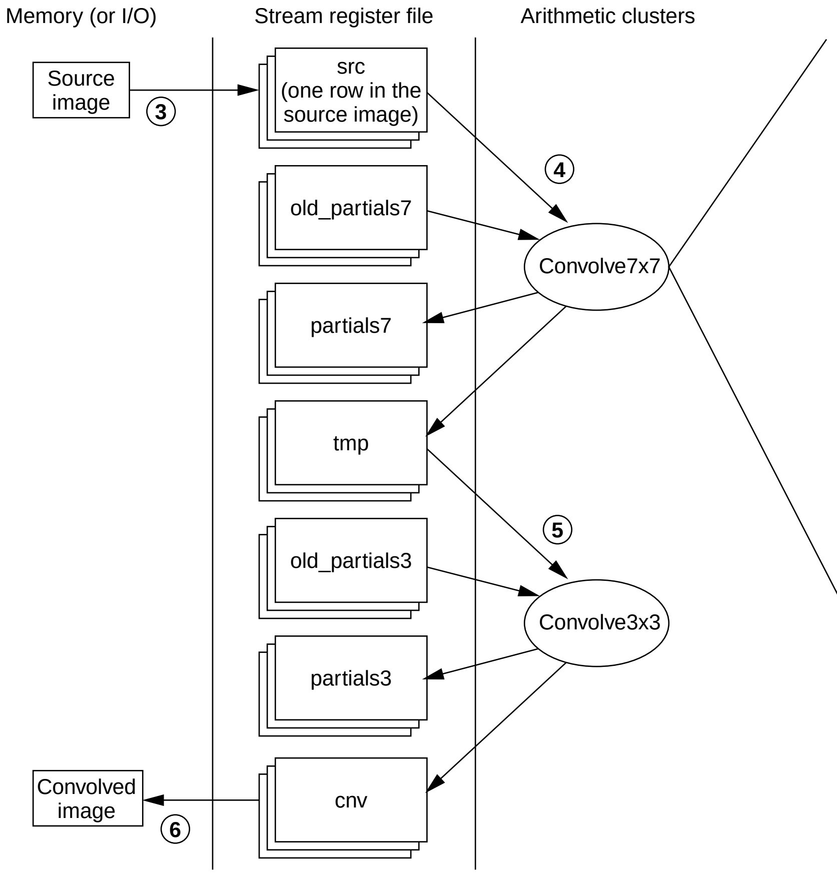
As the kernel executes, it loops through all input-stream elements. During each loop iteration, the clusters read eight new stream elements (one new element to each cluster) in parallel from the SRF into their LRFs. Then each cluster performs the same compound stream operation on its stream element.

In the convolve7×7 kernel, the compound stream operation must first perform six intercluster communications that broadcast the current data. As a result, the first cluster has the values $src[n-6: n]$, the second has $src[n-5: n+1]$, and so on. Edge clusters use data buffered from the previous iteration if necessary. Each cluster then multiplies these seven values with a $7 \times 7$ matrix of filter coefficients by executing seven dot products. The kernel uses the seven results to generate a new output element and to update the partial sums for future rows.

The convolve computation occurs in parallel on all eight arithmetic clusters. In each loop iteration, the LRFs temporarily store intermediate data between operations. At the end of the loop, the output stream elements are written back eight at a time (one from each cluster) to the SRF. In this manner, six values from each cluster are written to the partials7 output stream, and one new value per cluster is written to the tmp output stream. Upon completion, the SRF contains a stream con-

```
1  prime partials with arc_rows[0..5]
2  for (n = 6; n < numRows; n++){
3    src = load( src_rows[n] );                    //'src' stream gets one row of source image
4    convolve7x7( src, old_partials7, &tmp, &partials7 );
5    convolve3x3( tmp, old_partials3, &cnv, &partials3 );
6    convolved_rows[n-6 ] = store( cnv );             // store 'cnv' stream to memory
7    swap pointer to start of old_partials and partials for next time through the loop
8  }
9  drain partials to get convolved_rows[numRows-6..numRows-1]
```

**(a)**



```
convolve7x7(in, partials_in, out, partials_out){
:
:
while (! in.empty()){
  in >> curr[6]; // Input stream element

  // Communicate values to neighboring
  // clusters (edge clusters get buffered
  // data from prev iteration)
  for (i = 0; i < 6; i++)
    curr[i] = communicate(curr[6], perm[i])

  for (i = 0; i < 7; i++)
    rowsum[i] = dotproduct(curr, fltr[6-i]);

  partials_in >> p;
  out << p[0] + rowsum[0];
  for (i = 0; i < 5; i++)
    p[i] = p[i+1]+ rowsum [i+1];
  p[5] = r[6];
  partials_out << p;
 }
}
```

**(b)** **(c)**

Figure 3. The convolution stage of stereo depth extraction: application-level pseudocode (a); Imagine execution of pseudocode lines 3 through 6 (b); kernel-level pseudocode for 7 × 7 convolution (c).

sisting of a new row of convolved pixels (tmp) and a stream consisting of partial sums (partials7) for later use.

*Convolve3×3 kernel execution.* Now the stream controller issues the convolve3x3 kernel on line 5 of Figure 3a. The output stream of the previous convolve7x7 kernel is used as one of the input streams. Once the convolve3×3 kernel completes, the SRF contains the final convolved row of pixels.

*Storage.* Finally, line 6 transfers the final output stream to the memory system for storage and use in later stages of the application.

The rest of the stereo depth extraction application and other stream applications exe-

Table 1. Application bandwidth requirements versus Imagine's bandwidth hierarchy.

| | Memory (Gbytes/s) | SRF (Gbytes/s) | Clusters (Gbytes/s) | Performance |
|---|---|---|---|---|
| Imagine peak | 2.67 | 32 | 544 | 20 Gflops (40 16-bit GOPS) |
| Depth | 0.83 | 21.08 | 263.02 | 12.1 GOPS (16-bit) |
| MPEG2 | 0.45 | 2.21 | 214.31 | 18.3 GOPS (16- and 8-bit) |
| QR | 0.37 | 3.99 | 294.82 | 13.1 Gflops |
| Render | 1.61 | 8.59 | 205.05 | 5.1 GOPS (16-bit and floating-point) |

cute similarly. That is, kernels execute on the arithmetic clusters and streams pass between kernels through the SRF. In this manner, the stream programming model maps directly to the Imagine architecture. As a result, Imagine's 48 on-chip arithmetic units can take advantage of the high ratio of arithmetic operations to memory accesses. The SIMD nature of the arithmetic clusters and compound stream operations enables Imagine to exploit data parallelism.

Finally, multiprocessor solutions can take advantage of even more parallelism by using the network interface, which supports a peak bandwidth of 4 Gbytes per second in and out. For example, the stereo depth extraction application could use different Imagine processors to work on different portions of the image to be convolved. The application could exploit even more parallelism with control partitioning: One set of processors could work on the convolution stage and send the result streams to another set working on the SAD stage.

## Bandwidth hierarchy

The stream programming model also exposes the application's bandwidth requirements to the hardware. Imagine exploits this by providing a three-level bandwidth hierarchy:[2] off-chip memory bandwidth (2.67 Gbytes per second), SRF bandwidth (32 Gbytes/s), and intracluster bandwidth (544 Gbytes/s). The three levels of the bandwidth hierarchy correspond to the three columns of Figure 3b. Stream programs' communication patterns match this bandwidth hierarchy. Stream programs use memory bandwidth only for application input and output and when intermediate streams cannot fit in the SRF and must spill to memory. SRF bandwidth serves only when streams pass between kernels. Finally, intracluster bandwidth into and out of the LRFs handles the bulk of data during kernel execution.

Table 1 shows how four stream applications use Imagine's peak bandwidths. The four applications are the stereo depth extractor (Depth) described earlier, a video-encoding application (MPEG2), a QR matrix decomposition (QR), and polygon rendering on the SPECViewperf 6.1 ADVS-1 benchmark (Render). Each application shows roughly an order-of-magnitude increase in bandwidth requirements across each hierarchy level. By keeping the 48 arithmetic units supplied with data, the bandwidth hierarchy enables Imagine to sustain a large amount of its peak performance in these applications.

Imagine's bandwidth hierarchy also matches the capabilities of modern VLSI technology. The off-chip memory bandwidth is roughly the maximum processor-to-DRAM bandwidth available in today's technology. The SRF bandwidth is roughly the maximum bandwidth available from a single on-chip SRAM. The intracluster bandwidth is approximately the maximum bandwidth attainable from on-chip interconnect.

*Streaming memory system.* The memory system supports data stream transfers between the SRF and off-chip DRAM. Imagine makes all memory references using stream Load and Store instructions that transfer an entire stream between memory and the SRF. In the convolution stage example, stream elements were laid out sequentially in memory. However, the memory system also supports strided, indexed, and bit-reversed record addressing, for which records must be contiguous. In addition, because memory-reference granularity is streams, users can optimize the memory system for stream throughput rather than the reference latency of individual stream elements. A stream consumer, such as a kernel executing in the arithmetic clus-

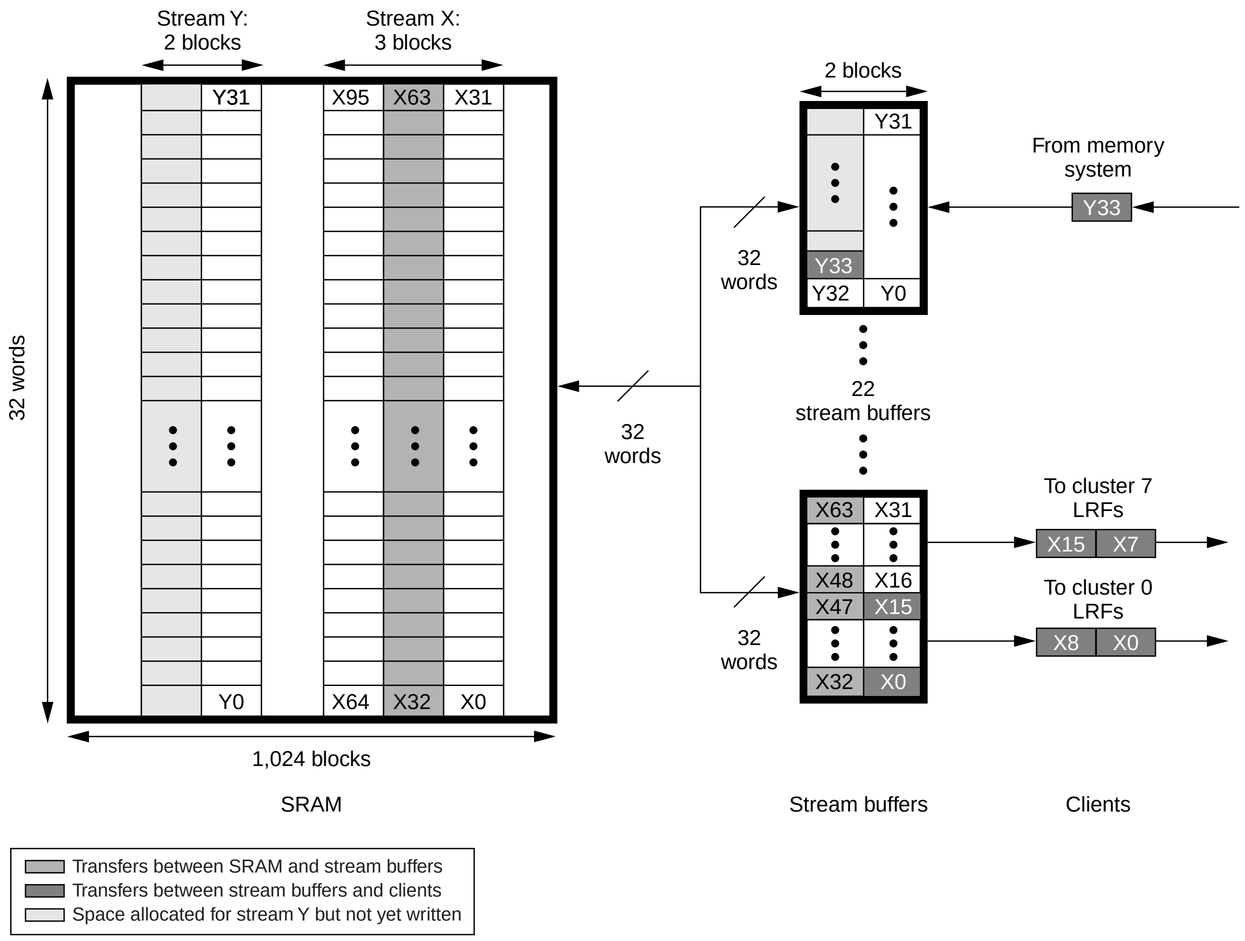


Figure 4. Stream register file organization.

ters, cannot begin until the entire stream is available in the SRF. Therefore, the time it takes to load the entire stream is far more important than the latency of any particular reference. Moreover, these references and computation can easily overlap. During the current set of computations, the memory system can load streams for the next set and store streams from the previous set.

Imagine uses memory access scheduling to reorder the DRAM operations (bank precharge, row activation, and column access) necessary to complete the set of currently pending memory references. When scheduling memory accesses, the memory system can delay references to let other references access the DRAM, improving memory bandwidth. Furthermore, a good reordering reduces the average latency of memory references by using the DRAM's internal resources more efficiently. Imagine's memory access scheduling has achieved a 30 percent performance improvement on a set of representative media-processing applications.[3]

*Stream register file.* The SRF contains a 128-Kbyte SRAM organized as 1,024 blocks of 32 words of 32 bits each. Figure 4 shows two concurrent SRF accesses. Stream Y (length 64 words) is being written from the memory system to the SRF, and stream X (length 96 words) is being read from the SRF by the arithmetic clusters. All client stream accesses go through stream buffers, which can hold two blocks of a stream each. Imagine has a total of 22 stream buffers, each with different-size ports:

- eight cluster stream buffers (eight words per cycle),

- eight network stream buffers (two words per cycle),
- four memory system stream buffers (one word per cycle),
- one microcontroller stream buffer (one word per cycle), and
- one stream buffer that interfaces to the host processor (one word per cycle).

Different clients have different numbers of stream buffers with various-size ports to provide the bandwidths and number of concurrent streams necessary for a typical application.

During execution, active stream buffers contend for access to a SRAM block. In Figure 4, the second block of stream X is currently being read from the SRAM and written into the bottom stream buffer. The clusters read eight words out of their stream buffers at a time (one word to each cluster). In this example, the clusters have already completed two reads (X0-X7 and X8-X15) on previous cycles from the bottom stream buffer. Once the clusters have read elements X16-X31, the third block of stream X (X64-X95) replaces X0-X31 in the stream buffer.

Concurrently, the memory system is writing element Y33 of stream Y into its stream buffer. Once elements Y34-Y63 are written into the stream buffer, the second block of stream Y (Y32-Y63) is transferred to the SRAM, and the SRF stream write ends. The first block of the stream (Y0-Y31) was transferred into the SRF on a previous cycle.

Because all data and accesses to the SRF are organized as streams, stream buffers can take advantage of the sequential access pattern of streams. Stream buffers prefetch data one block at a time from the SRF, while clients read data from stream buffers at lower bandwidths. As a result, stream buffers effectively time-multiplex the single physical port of the SRF SRAM into 22 logical ports that can be accessed simultaneously. In contrast with building large multiported structures to provide the necessary bandwidth, stream buffers lead to an area- and power-efficient VLSI implementation.[4] Moreover, they provide a simple, extensible mechanism by which clients can execute stream transfers with the SRF.

*Arithmetic clusters.* Each arithmetic cluster contains eight functional units: three adders, two multipliers, a divide and square-root unit, a scratch-pad register file unit, and an intercluster communication unit. The adders and multipliers execute floating-point and integer arithmetic with a throughput of one instruction per cycle at varying latencies. They also support parallel subword operations such as those in MMX[5] and other multimedia extensions.[6] The divide and square-root unit executes floating-point and integer operations and is pipelined to accept two instructions every 13 cycles. The scratch-pad unit is a 256-word register file that executes indexed read and write instructions useful for small table lookups. The intercluster communication unit, a software-controlled crossbar between the clusters, is used when kernels are not fully data parallel. The communication unit is also used by conditional streams,[7] a mechanism for handling data-dependent conditionals in a stream architecture.

In front of each functional unit is a small, 16-word, two-ported LRF. (Multipliers have 32-word LRFs because many kernels require a larger number of registers in the multiplier than in other units.) The intracluster switch transfers data between functional units or cluster stream buffers and LRFs. The switch consists of a multiplexer in front of every LRF input; the multiplexer selects a functional unit or cluster stream buffer output to write from.

The LRFs are a much more area- and power-efficient structure for providing local bandwidth and storage than the equivalent multiported global register file. A multiported register file has, in effect, an implicit switch that is replicated many times inside each register cell. The distributed LRFs replace this structure with an explicit switch outside the LRFs.

## Evaluation

The Imagine stream architecture directly runs media applications written in the stream programming model and expressed in two programming languages: StreamC and KernelC. A StreamC program describes the interactions of streams and kernels and corresponds to the pseudocode in Figure 3a. A kernel is written in KernelC and corresponds to the pseudocode in Figure 3c. Both languages use C-like syntax. StreamC is compiled by a stream scheduler, which handles SRF block allocation and memory manage-

## Imagine and related work

Prior industry and research media processors fall into four categories:

- VLIW media processors and DSPs,
- SIMD extensions,
- hardwired stream processors, and
- vector processors.

VLIW DSPs, such as the Texas Instruments C6x,[1] and VLIW media processors, such as Trimedia,[2] efficiently exploit the instruction-level parallelism in media applications. SIMD extensions of instruction sets enable many general-purpose processors[3,4] to perform low-precision operations and exploit fine-grained data parallelism.

Imagine's VLIW arithmetic clusters and subword arithmetic operations build on this previous work. However, the stream architecture enables Imagine to scale to much larger numbers of ALUs. Moreover, because both the VLIW and the SIMD approaches focus on execution unit architecture, neither addresses the data bandwidth problem solved by Imagine's register hierarchy.

Hardwired stream processors attempt to map the stream programming model directly into hardware. Cheops,[5] for example, consists of a set of specialized stream processors. Each processor accepts one or two data streams as input and produces one or two data streams as output. Data streams are either forwarded directly from one stream processor to the next according to the application's dataflow graph or transferred between memory and the stream processors.

Programmable stream processors like Imagine are closest in spirit to vector processors. Vector supercomputers,[6] SIMD processor arrays, and vector microprocessors such as the T0,[7] Vector IRAM,[8] and others efficiently use vectors to exploit data parallelism. Vector memory systems are suitable for media processing because they are optimized for bandwidth instead of latency. Vector processors perform simple arithmetic operations on vectors stored in a vector register file, using three words of vector register bandwidth per arithmetic operation.

In contrast, Imagine performs compound stream operations, using local register files to stage intermediate results. This reduces bandwidth demand on the stream register file by an order of magnitude or more. Imagine also differs from vector processors by using stream buffers to take advantage of the sequential data access patterns of streams and by supporting data records. Finally, vector processors execute instructions from one sequencer that manages tightly coupled vector and scalar units. In contrast, Imagine works directly on the stream programming model. Therefore, Imagine maintains an explicit division between stream-sequencing instructions (written in StreamC) executed on the host processor and VLIW instructions (written in KernelC) executed on Imagine's arithmetic clusters.

### References

1. T. Halfhill, "TI Cores Accelerate DSP Arms Race," *Microprocessor Report,* Mar. 2000, pp. 22-28.
2. S. Rathnam and G. Slavenburg, "An Architectural Overview of the Programmable Multimedia Processor, TM-1," *Proc. Compcon*, IEEE Computer Society Press, Los Alamitos, Calif., 1996, pp. 319-326.
3. A. Peleg and U. Weiser, "MMX Technology Extension to the Intel Architecture," *IEEE Micro*, vol. 17, no. 4, Aug. 1996, pp. 42-50.
4. M. Tremblay et al., "VIS Speeds New Media Processing," *IEEE Micro*, vol. 16, no. 4, Aug. 1996, pp. 10-20.
5. V.M. Bove Jr. and J.A. Watlington, "Cheops: A Reconfigurable Data-Flow System for Video Processing," *IEEE Trans. Circuits and Systems for Video Technology*, vol. 5, no. 2, Apr. 1995, pp. 140-149.
6. R. Russell, "The Cray-1 Computer System," *Comm. ACM*, vol. 21, no. 1, Jan. 1978, pp. 63-72.
7. J. Wawrzynek et al., "Spert-II: A Vector Microprocessor System", *Computer,* vol. 29, no. 3, Mar. 1996, pp. 79-86.
8. C. Kozyrakis, *A Media-Enhanced Vector Architecture for Embedded Memory Systems,* tech. report UCB/CSD-99-1059, Univ. of California at Berkeley, Computer Science Div., Berkeley, Calif., 1999.

ment. The stream scheduler runs on the host processor and uses a combination of static and runtime techniques to send stream instructions to the Imagine processor's stream controller. KernelC is compiled statically by an optimizing VLIW kernel scheduler that handles the communication required by the distributed register file architecture of the arithmetic clusters.[8]

Using these programming tools, we wrote kernels and applications for the Imagine processor. We simulated the four applications listed in Table 2 (next page) on a cycle-accurate C++ simulator, assuming a 500-MHz cycle time and a 167-MHz SDRAM. Individual kernels are usually part of a larger application, so we simulated the kernels with their working sets already in the SRF.

Table 2 shows that Imagine sustained over half its peak performance on a range of applications and kernels. The performance differences between kernels and applications reflect the overhead of accessing off-chip memory and the start-up cost of stream transfers. An example is the performance degradation between the discrete cosine transform kernel and the rest of the MPEG-2 encoding application.

Table 2. Application and kernel performance.

| | Arithmetic bandwidth | Power estimate (W) | Application performance |
|---|---|---|---|
| **Applications** | | | |
| Depth | 12.1 GOPS (16-bit) | 2.9 | 320 × 240 8-bit gray scale at 212 frames/s |
| MPEG2 | 18.3 GOPS (16- and 8-bit) | 2.2 | 720 × 480 24-bit color at 105 frames/s |
| QR | 13.1 Gflops | 3.6 | 192 × 96 matrix decomposition in 1.1 ms |
| Render | 5.1 GOPS (16-bit and floating-point) | 2.9 | 14.9 million vertices/s (16.8 million pixels/s) |
| **Kernels** | | | |
| Discrete cosine transform | 22.6 GOPS (16-bit) | 2.6 | 34.8 ns per 8 × 8 block (16-bit) |
| Convolve7×7 | 25.6 GOPS (16-bit) | 3.0 | 1.5 μs per row of 320 16-bit pixels |
| Fast Fourier transform | 6.9 Gflops | 3.8 | 7.4 μs per 1,024-point floating-point complex FFT |

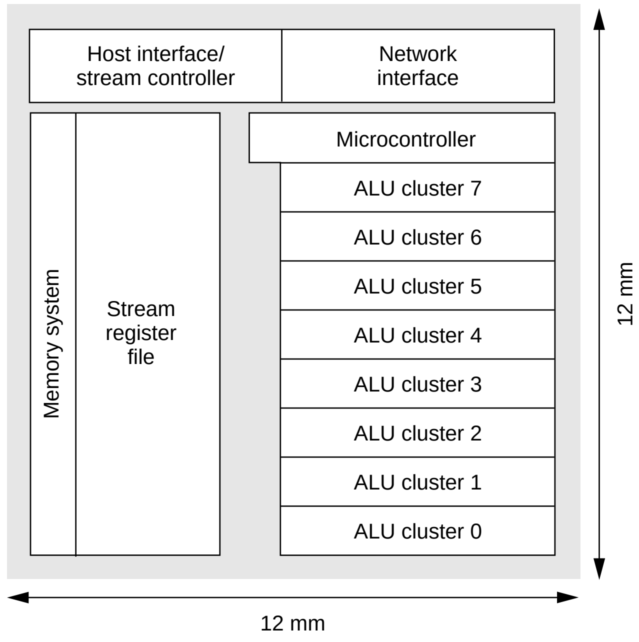


Figure 5. Floor plan of the Imagine prototype.

Table 2 also shows Imagine's estimated power dissipation. We used trial layouts, netlists, and SRAM data sheets in conjunction with average unit occupancies from the cycle-accurate simulator to estimate the total capacitance switched at runtime. We expect Imagine to dissipate less than 4 W over a range of applications, providing power efficiencies of more than 2 Gflops/W. Imagine's power efficiency stems from its efficient register file organization. For architectures with many ALUs, the SRF and LRF structures dissipate much less power in inter-ALU communications than do conventional register file organizations.[4]

Imagine's performance and power efficiency across a range of applications are roughly an order of magnitude higher than those of today's programmable processors such as the Texas Instruments TMS320C67x. For comparison, a 167-MHz TI TMS320C6701 executes a 1,024-point floating-point complex FFT in 124.3 μs with an estimated 1.9-W power dissipation (0.41 Gflops and 0.22 Gflops/W).[9] Imagine's performance is also comparable to that of special-purpose processors on complex applications such as polygon rendering. A study compared Imagine polygon-rendering simulations with a 120-MHz NVIDIA Quadro with DDR SDRAM on an Intel AGP 4X system.[10] In rendering a rasterization-limited scene, the NVIDIA Quadro provided a 5.3-times speedup over Imagine. For geometry-limited scenes (such as the SPECViewperf 6.1 ADVS-1 polygon-rendering scene with point-sampled texture listed in Table 2), Imagine outperformed the NVIDIA Quadro.

## VLSI implementation

We are currently designing a prototype Imagine stream processor. Our goal is to validate architectural studies and provide an experimental prototype for future research. The processor consists of approximately 21 million transistors: 6 million in the SRF SRAM, 6 million in the microcode store, 6 million in the arithmetic clusters, and 3 million elsewhere. We have targeted the prototype to operate at 500 MHz in a 0.15-μm, 1.5-V static CMOS standard-cell technology. As Figure 5 shows, Imagine has a die size of 1.44 cm.[2] It is also area-efficient. Nearly 40 percent of the die area is devoted to the arithmetic clusters.

In addition to providing high performance density, the Imagine stream architecture meets the challenges of modern VLSI design complexity. The architecture's regularity lets designers heavily optimize one module and then instantiate it in many parts of the design. The statically scheduled arithmetic clusters contain explicit communication with only local data forwarding, eliminating the need for complicated global control structures. Data and control communication between different storage hierarchy levels (LRFs, SRF, and streaming memory system) is local, mitigating the effect of long wire delays.

The Imagine design team consists of eight graduate students. Two to three people work on architecture and logic design, two to three on software tools, and the others on VLSI implementation. With this limited staff, we need a CAD flow that minimizes design time with minimal performance degradation. We use a typical application-specific integrated circuit synthesis flow with automatic placement and routing for control blocks. For regular data path blocks such as the multiplier array, we avoid synthesis, instead using data-path-style placement of standard cells with automatic routing. We synthesize other, less regular blocks, such as the floating-point adder, and use data-path-style standard-cell placement. This CAD flow provides a good trade-off between design time and performance and helps the team build a high-performance processor in spite of the limited resources.

We completed many design tools and studies to verify Imagine's performance and VLSI feasibility. We developed a cycle-accurate C++ simulator, which we used to generate all performance numbers given here. We also completed a synthesizable register-transfer level model of Imagine and mapped the design to logic gates. Placement and routing are in progress, and we expect to have prototype processors available in August 2001.

To gain more experience with stream programming, we plan to build Imagine systems ranging from a single-processor, 20-Gflops system to a 64-processor, Tflops system for demanding applications.

Stream processing is a promising technology now in its infancy. Our future research will focus on generalizing the stream model for a wider range of applications and refining the stream processor architecture. MICRO

## Acknowledgments

We thank all the past and present members of the Imagine team. We also thank our industrial partners Intel and Texas Instruments. Our research was supported by the Defense Advanced Research Projects Agency under ARPA order E254 and monitored by the Army Intelligence Center under contract DABT63-96-C-0037. Several of the authors received support from the National Science Foundation and/or Stanford graduate fellowships.

## References

1. T. Kanade et al., "Development of a Video-Rate Stereo Machine," *Proc. Int'l Robotics and Systems Conf.,* IEEE Press, Piscataway, N.J., 1995, pp. 95-100.
2. S. Rixner et al., "A Bandwidth-Efficient Architecture for Media Processing," *Proc. 31st Int'l Symp. Microarchitecture*, IEEE Computer Society Press, Los Alamitos, Calif., 1998, pp. 3-13.
3. S. Rixner et al., "Memory Access Scheduling," *Proc. 27th Ann. Int'l Symp. Computer Architecture*, IEEE CS Press, 2000, pp. 128-138.
4. S. Rixner et al., "Register Organization for Media Processing," *Proc. Sixth Int'l Symp. High-Performance Computer Architecture*, IEEE CS Press, 2000, pp. 375-387.
5. A. Peleg and U. Weiser, "MMX Technology Extension to the Intel Architecture," *IEEE Micro*, vol. 17, no. 4, Aug. 1996, pp. 42-50.
6. M. Tremblay et al., "VIS Speeds New Media Processing," *IEEE Micro*, vol. 16, no. 4, Aug. 1996, pp. 10-20.
7. U. Kapasi et al., "Efficient Conditional Operations for Data-Parallel Architectures," *Proc. 33rd Int'l Symp. Microarchitecture*, IEEE CS Press, 2000, pp. 159-170.
8. P. Mattson et al., "Communication Scheduling," *Proc. Ninth Int'l Conf. Architectural Support for Programming Languages and Operating Systems*, ACM Press, New York, 2000, pp. 82-92.
9. T. Halfhill, "TI Cores Accelerate DSP Arms Race," *Microprocessor Report*, Mar. 2000, pp. 22-28.
10. J. Owens et al., "Polygon Rendering on a Stream Architecture," *Proc. SIGGRAPH/*

*Eurographics Workshop on Graphics Hardware*, ACM Press, New York, 2000, pp. 23-32.

**Brucek Khailany** is a PhD candidate in the Computer Systems Laboratory at Stanford University. He leads the VLSI implementation of the Imagine processor. Khailany has an MS in electrical engineering from Stanford University and a BS in electrical engineering from the University of Michigan. He is a student member of the IEEE and the ACM.

**William J. Dally** is a professor of electrical engineering and computer science at Stanford University, where his group developed the Imagine processor. He currently leads projects on high-speed signaling, computer architecture, and network architecture. Dally has a BS in electrical engineering from Virginia Polytechnic Institute, an MS in electrical engineering from Stanford University, and a PhD in computer science from Caltech. He is a member of the Computer Society, the Solid-State Circuits Society, the IEEE, and the ACM.

**Ujval J. Kapasi** is a PhD candidate in the Department of Electrical Engineering at Stanford University. He is one of the architects of the Imagine stream processor, and his research interests lie in parallel computer systems and software support for these systems. Kapasi holds an MS from Stanford University and a BS from Brown University.

**Peter Mattson** is a PhD candidate at Stanford University. His research interests include programming languages and compilers for high-performance processors. He has a BS in electrical engineering from the University of Washington and an MS in electrical engineering from Stanford University. He is a member of the IEEE and the ACM.

**Jinyung Namkoong** is working toward the PhD in electrical engineering at Stanford University. His research interests include VLSI circuit techniques, systems architecture, and interconnection networks. Namkoong has a BS in electrical engineering and computer science from the University of California at Berkeley and an MS in electrical engineering from Stanford University.

**John D. Owens** is a PhD candidate in electrical engineering at Stanford University, where he is an architect of the Imagine stream processor. His research interests are computer architecture and computer graphics. Owens has a BS in electrical engineering and computer science from the University of California at Berkeley and an MS in electrical engineering from Stanford.

**Brian Towles** is a master's student in electrical engineering at Stanford. He is a system architect for the Imagine processor, with research interests in computer architecture and on-chip interconnection networks. Towles has a BS in computer engineering from the Georgia Institute of Technology.

**Andrew Chang** is a PhD candidate in electrical engineering at Stanford. His research interests include VLSI CAD, computer architecture, and circuit design. Chang received a BS and an MS degrees from the Massachusetts Institute of Technology. He is a student member of the IEEE and the ACM.

**Scott Rixner** is an assistant professor of computer science and electrical and computer engineering at Rice University. Previously, he was the lead architect for the Imagine stream processor. His interests include computer architecture and media processing. Rixner has a BS and an MS in computer science and engineering and a PhD in electrical engineering from the Massachusetts Institute of Technology. He is a member of Tau Beta Pi and the ACM.

Direct questions and comments about this article to Brucek Khailany, Stanford University, Computer Systems Laboratory, Stanford, CA 94305; khailany@cva.stanford.edu.