Appears in Micro-31 (1998)

A Bandwidth-Efficient Architecture for Media Processing

Scott Rixner[1], William J. Dally, Ujval J. Kapasi, Brucek Khailany,
Abelardo López-Lagunas, Peter R. Mattson, and John D. Owens

Computer Systems Laboratory
Stanford University
Stanford, CA 94305
{rixner, billd, ujk, khailany, alopez, pmattson, jowens}@cva.stanford.edu

Abstract

Media applications are characterized by large amounts of available parallelism, little data reuse, and a high computation to memory access ratio. While these characteristics are poorly matched to conventional microprocessor architectures, they are a good fit for modern VLSI technology with its high arithmetic capacity but limited global bandwidth. The stream programming model, in which an application is coded as streams of data records passing through computation kernels, exposes both parallelism and locality in media applications that can be exploited by VLSI architectures. The Imagine architecture supports the stream programming model by providing a bandwidth hierarchy tailored to the demands of media applications. Compared to a conventional scalar processor, Imagine reduces the global register and memory bandwidth required by typical applications by factors of 13 and 21 respectively. This bandwidth efficiency enables a single chip Imagine processor to achieve a peak performance of 16.2GFLOPS (single-precision floating point) and sustained performance of up to 8.5GFLOPS on media processing kernels.

1. Introduction

Application and technology trends together motivate a departure from the scalar, general-purpose register architecture in wide use today toward a stream-based architecture with a bandwidth-efficient register organization. Media applications are already a dominant consumer of computing cycles and are projected to account for over 90% of the cycles consumed by the year 2000 [2] [3] [6]. These applications, including rendering 2-D and 3-D graphics, image and audio compression and decompression, and image processing, operate on large *streams* of low-precision integer data and share three key characteristics. First, operations on one stream element are largely independent of the others. Thus, they can exploit large amounts of parallelism and tolerate large amounts of latency. Second, every stream element is read exactly once, resulting in poor cache performance. Finally, they are computationally intensive, often performing 100-200 arithmetic operations for each element read from memory. These applications are poorly matched to conventional architectures that cannot exploit the available parallelism, are optimized for low latency, depend on data reuse, and cannot support a high computation to memory access ratio. These applications are well matched, however, to the characteristics of modern VLSI technology.

Modern VLSI computing systems are limited by communication bandwidth rather than arithmetic. In a contemporary 0.25μm CMOS technology, a 32-bit adder requires less than 0.25mm^2 of chip area and a multiplier is smaller than 0.5mm^2 of area.[2] Hundreds of these arithmetic units fit on an inexpensive 1cm^2 chip. The challenge is keeping these hungry units fed with instructions and data. It is infeasible to provide the data bandwidth required out of a global register file or the instruction bandwidth needed from a global issue unit. Locality is required to realize the potential of the technology. Fortunately, the streaming nature of media applications provides exactly the locality

[1] Scott Rixner is an Electrical Engineering graduate student at the Massachusetts Institute of Technology.

The research described in this paper 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.

[2] Based on area measurements taken from automatically generated layouts of actual arithmetic units.

needed. Forwarding streams of data from one processing kernel to the next localizes data communication and makes it easy to manage. Exploiting data parallelism allows a single instruction to be used by multiple arithmetic units and localizes data to a small cluster of units. The computational intensity of the applications can be exploited through instruction-level parallelism. Most importantly, it is easy to program a media application as a sequence of operations on streams of data in a manner that exposes the parallelism and locality required to execute the algorithm on a VLSI architecture.

The Imagine architecture matches the demands of media applications to the capabilities of VLSI technology by supporting a stream-based programming model.[3] Imagine is organized around a large (64KB) stream register file (SRF). Load and store operations move entire streams of data between memory and the SRF. To the programmer, Imagine is a load/store architecture for streams: one codes an application to load streams into the SRF, pass these streams through a number of computation kernels, and store the results back to memory.

A stream computation, for example transforming triangle vertices, is performed by reading a stream from the SRF, passing its elements through a set of eight arithmetic clusters, and storing the results back into the SRF. Both data and instruction-level parallelism are exploited in a stream computation. The arithmetic clusters work in parallel on different elements of the stream and each cluster has several arithmetic units that operate under VLIW control on a single data element. Intermediate results during a computation are kept local to a cluster and do not use SRF bandwidth. This allows data bandwidth to be used efficiently in the sense that expensive, communication limited global register bandwidth is not wasted on the arithmetic units where inexpensive local bandwidth is easy to provide and use. Similarly, the recirculation of streams through the large SRF minimizes the use of scarce off-chip data bandwidth in favor of global register bandwidth. This is in contrast to conventional architectures which use less efficient global register bandwidth when local bandwidth would suffice, in turn forcing the use of more off-chip bandwidth.

This paper introduces *streams* as a programming model and describes how the Imagine architecture uses a storage bandwidth hierarchy to exploit the parallelism and locality of streaming applications and achieve very high performance in a single-chip media processor. As described in Section 2, media applications are easily expressed as a sequence of computation kernels that operate on streams of data. Triangle rendering, for example, can be expressed as passing a stream of triangles through the stages of the traditional graphics pipeline. The Imagine architecture, described in Section 3, provides a three-level storage hierarchy designed to support streaming applications. Programming of Imagine by writing a set of kernels that are then sequenced by application-level instructions is described in Section 4. The performance and bandwidth demands of applications are discussed in Section 5, and the stream architecture is compared to alternate architectures.

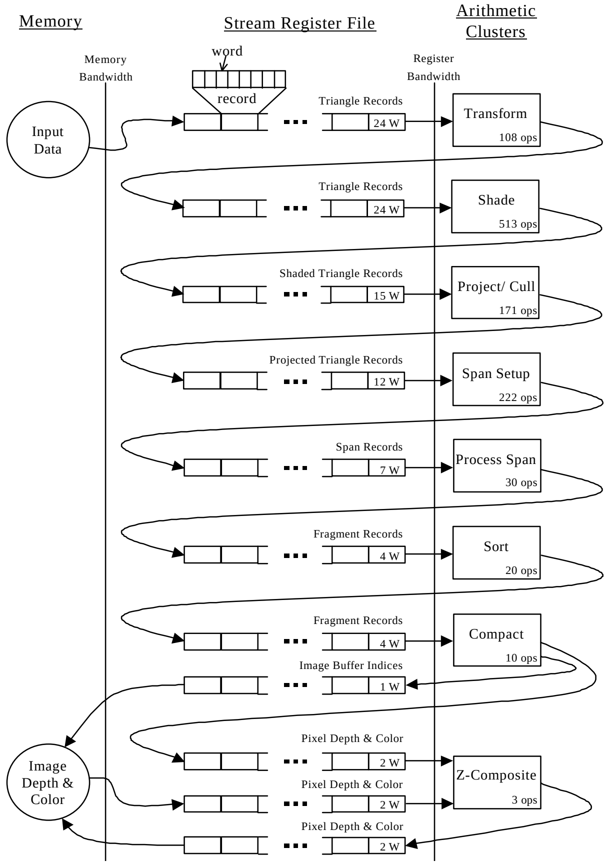

Figure 1: Stream-Based Triangle Rendering

2. Stream Processing

Media processing applications are easily expressed as a series of computation kernels that operate on large data streams. A kernel is a small program that is repeated for each successive element of its input streams to produce output streams for the next kernel in the application. Streams are organized as a sequence of records. Each record in a stream is a collection of related data words. Expressing a computation in the stream model enables a system to meet the large instruction and data bandwidth demands of media processing. The data bandwidth required by the kernels can be provided by a storage hierarchy optimized for streams, and the instruction bandwidth

[3] The Cheops video processor [1] was also organized around the concept of streams, but with specialized function units and without the memory hierarchy of Imagine.

requirements are greatly reduced by exploiting data parallelism across the records of a stream.

Figure 1 shows one way that triangle rendering [4], a representative media application, maps onto the stream processing model. Triangle rendering is composed of eight computation kernels that operate on data streams in succession. The input to the application is a stream of triangles used to represent objects. Each triangle in this stream is transformed into the viewing coordinate system, creating a new stream of triangles. This new stream is passed through the Phong shading kernel (although any shading algorithm could be used) to light the triangle vertices. This creates another stream, which is passed along to the next kernel, and so on, as depicted in Figure 1. The output of the application is the pixel color and depth information that is emitted from the z-composite kernel. As triangle streams are passed through the application, the image buffer will be the composition of all of the visible rendered pixels from those triangle streams. Ultimately, all of the objects' triangles will have passed through the computation pipeline, and the image buffer will hold the final rendered image.

The computation kernels themselves are expressed as *compound stream operations*. A compound stream operation is a small program that has access to the record at the head of each of its input streams and to its local variables. Explicit instructions read the input streams, so there is no need for the consumption rate of the input streams to be matched. Similarly, the tail of each output stream can be written explicitly at any point in the program. Since all streams are read and written independently, the length and record size of each stream can be different. These operations are *compound* in the sense that they perform multiple arithmetic operations on each stream element. This is in contrast to conventional vector (or stream) operations that perform a single operation on each element of the vector.

Consider, for example, the first kernel in triangle rendering: model to world space transformation. For this transformation, there is only a single input and a single output stream. Both streams consist of 24 element triangle records, with the input triangles positioned in model space, and with the output triangles positioned in world space. Each triangle is composed of three vertices. A vertex contains eight 32-bit words: its three-dimensional coordinates in single precision floating point, a homogenous coordinate used for perspective, its *rgb* color packed into a single 32-bit integer, and the normal vector for the vertex described by three coordinates in single precision floating point. The transformation computation can be expressed as a single compound stream operation, as shown in Figure 2. The outer loop is repeated for each 24 element triangle record in the input stream.

```
loop over all triangles {
  loop over 3 vertices {
    // read vertex data from input stream
    [x, y, z, w, rgb, nx, ny, nz] = InputStream0;

    // compute transformed vertex coordinates
    tx = r11 * x + r12 * y + r13 * z + r14 * w;
    ty = r21 * x + r22 * y + r23 * z + r24 * w;
    tz = r31 * x + r32 * y + r33 * z + r34 * w;

    // compute transformed normal vector
    tnx = n11 * nx + n12 * ny + n13 * nz;
    tny = n21 * nx + n22 * ny + n23 * nz;
    tnz = n31 * nx + n32 * ny + n33 * nz;

    // write vertex data to output stream
    OutputStream0 = [tx, ty, tz, w, rgb, tnx, tny, tnz];
  }
}
```

Figure 2: Transformation Kernel

Figure 1 also shows how triangle rendering maps onto a stream storage hierarchy. The storage hierarchy has three components: the memory system, the stream register file, and local register files in the arithmetic clusters. The off-chip memory holds persistent data. The SRF stores streams as they pass between computation kernels. The arithmetic clusters execute compound stream operations and contain local register files so that intermediate results do not need to recirculate through the stream register file.

The triangle rendering computation begins by reading a stream of triangles from memory to the stream register file as shown in the upper left corner of Figure 1. The application then proceeds vertically down the figure by passing streams through successive computation kernels, as described earlier. During the course of these computations, streams are recirculated through the stream register file, and do not need to return to memory. Intermediate data within the kernels are held in local registers and thus do not consume stream register bandwidth. Finally, after processing by eight kernels, a stream of pixels is written from the stream register file back to memory. For each step, the number associated with each stream in the stream register file represents the number of words per data record, and the number associated with each kernel denotes the number of arithmetic operations performed by that kernel on each input record.

For typical data (average triangles covering 25 pixels with a depth complexity of 5), rendering each triangle requires 1929 arithmetic operations, 666 SRF references, and 44 memory references. With a conventional microprocessor architecture, at least 5787 global register file references would be required (3 for each arithmetic operation). Thus, by capturing locality within the kernels, coding the application in the stream model reduces global register bandwidth demand by a factor of eight.

3. Imagine Stream Architecture

Imagine is a programmable single-chip processor that supports the stream programming model. Imagine provides

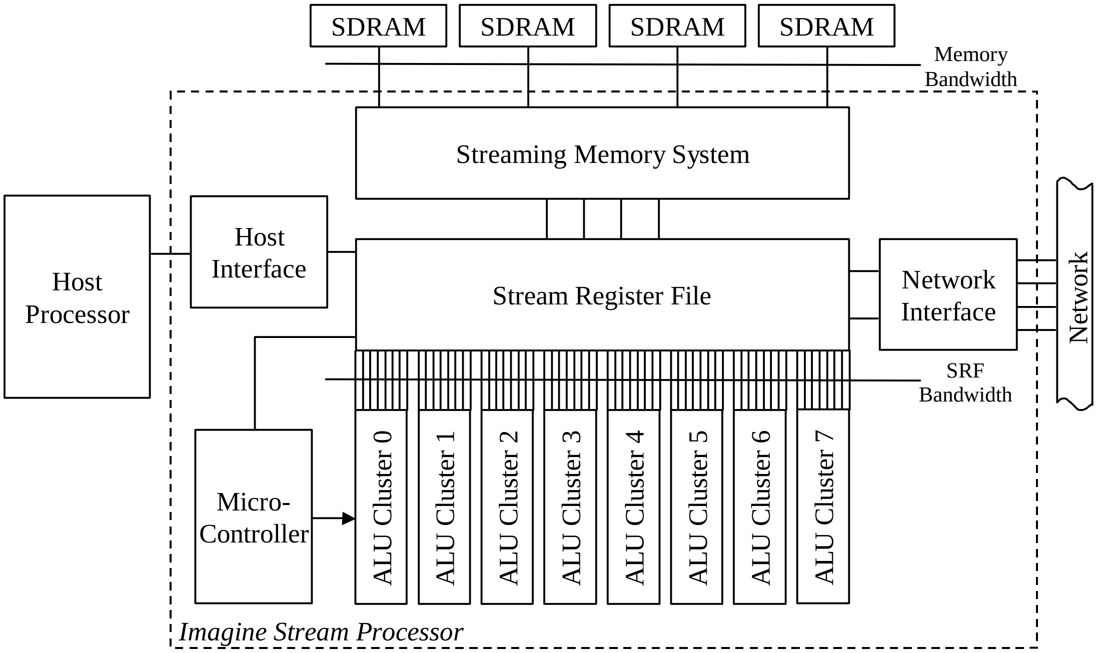

Figure 3: Block Diagram of Imagine

a storage bandwidth hierarchy that corresponds to the three levels of Figure 1. Using this hierarchy to exploit the parallelism and locality of streaming media applications, Imagine is able to sustain performance of 8.5GFLOPS on key kernels. This is comparable to special purpose processors, yet Imagine is still easily programmable for a wide range of applications. Imagine is designed to fit on a 1cm^2 0.25µm CMOS chip and to operate at 400MHz.

Imagine is a coprocessor that is programmed at two levels: kernel and application. Kernels, like the triangle transformation kernel in Figure 2, are coded in a programming language using the expression syntax of the C language. Kernels may access local variables, read input streams, and write output streams, but may not make arbitrary memory references. As described in Section 4, kernels are compiled into microcode programs that sequence the units within the arithmetic clusters to carry out the kernel function on each stream element in turn.

`Load`	*`Stream AddressDescriptor`*
`Store`	*`Stream AddressDescriptor`*
`Send`	*`Stream RoutingHeader Channel`*
`Receive`	*`Stream Channel`*
`Operate`	*`Kernel IStream0..IStream3 OStream0..OStream3`*

Figure 4: Imagine Application-Level Instruction Set

At the application level, Imagine is programmed using C++ library calls on a host processor. These library calls pass instructions to Imagine using the stream instruction set shown in Figure 4. Load and store instructions move streams between the SRF and memory. These instructions take a stream descriptor that identifies a starting location, length, and record size of a stream in the SRF, and an address descriptor that provides the base address in memory and addressing mode (constant stride, indexed, or bit-reversed). Send and receive instructions allow streams to be passed from the SRF of one Imagine to the SRF of another for multiprocessor applications. Finally, kernels are invoked by operate instructions. This instruction specifies the start address of the kernel in the control store of the microcontroller, stream descriptors for up to four source streams, and stream descriptors for up to four destination streams.

For example, the triangle rendering application of Figure 1 is coded with just 11 application-level instructions. One load instruction reads the stream of triangles from memory. Seven operate instructions sequence the kernels from TRANSFORM to COMPACT. A load instruction uses the index vector computed by COMPACT to read the old Z-values of the pixels in question. One more operate instruction initiates the Z-COMPOSITE kernel. Finally, a store instruction writes the visible pixels, and their Z-values, back to memory.

Figure 3 shows a block diagram of the Imagine stream processor. The stream register file (SRF) is the nexus of the processor. The memory system, arithmetic clusters, host interface, microcontroller, and network interface all interact by transferring streams to and from the SRF. Kernel

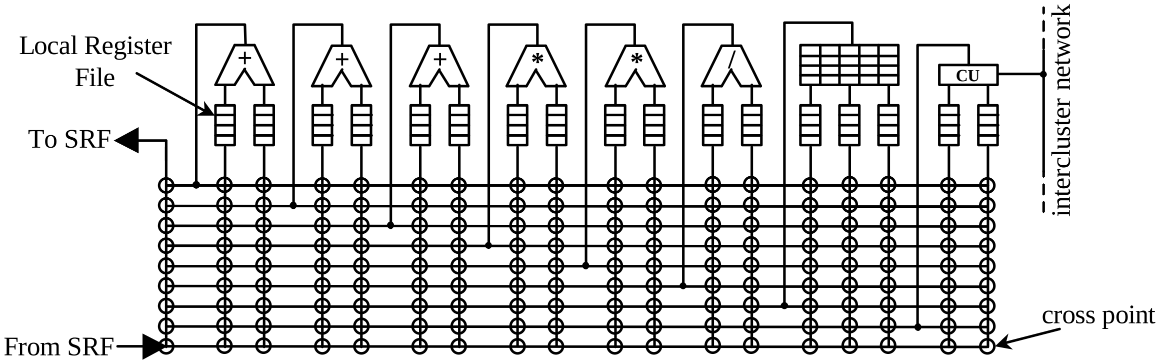

Figure 5: Imagine Cluster Organization

programs are loaded into the microcontroller's control store by loading streams from the SRF.

3.1 Stream Register File (SRF)

The SRF is a 64KB memory organized to handle streams. The SRF can hold any number of streams of any length. The only limitation is the actual size of the SRF. Streams are referenced using a *stream descriptor*, which includes a base address in the SRF, a stream length, and the record size of data elements in the stream.

An array of 18 64-word stream buffers is used to allow read or write access to 18 stream clients simultaneously. The clients are the units which access streams out of the SRF, such as the memory system, network interface, and arithmetic clusters. The internal memory array is 32 words wide, allowing it to fill or drain half of one stream buffer each cycle, providing a total bandwidth of 51.2GB/s for all 18 streams.

Each stream client may access its dedicated stream buffer every cycle if there is data available to be read or space available to be written. The eight stream buffers serving the clusters are accessed eight words at a time, one word per cluster, while the other ten stream buffers are accessed a single word at a time. The peak bandwidth of the stream buffers is therefore 74 words per cycle, or 118GB/s, allowing peak stream demand to exceed the SRF bandwidth during short transients. Stream buffers are bidirectional, but may only be used in a single direction for the duration of each logical stream transfer.

3.2 Memory System

As described above, all Imagine memory references are made using stream load and store instructions that transfer an entire stream between memory and the SRF. This stream load/store architecture is similar in concept to the scalar load/store architecture of contemporary RISC processors. It simplifies programming and allows the memory system to be optimized for stream throughput, rather than the throughput of individual, independent accesses. The memory system provides 1.6GB/s of bandwidth to off-chip SDRAM storage via four independent 32-bit wide SDRAM banks operating at 100MHz.[4] The system can perform two simultaneous stream memory transfers. To support these simultaneous transfers, four streams (two index streams and two data streams) connect the memory system to the SRF. Imagine addressing modes support sequential, constant stride, indexed (scatter/gather), and bit-reversed accesses on a record-by-record basis.

3.3 Cluster Array

Eight arithmetic clusters, controlled by a single microcontroller, perform kernel computations on streams of data. Each cluster operates on one record of a stream so that eight records are processed simultaneously. As shown in Figure 5, each cluster includes three adders, two multipliers, one divide/square root unit, one 128-entry scratch-pad register file, and one intercluster communication unit. This mix of arithmetic units is well suited to our experimental kernels. However, the architectural concept is compatible with other mixes and types of arithmetic units within the clusters.

Each input of every functional unit in the cluster is fed by a separate sixteen element local register file (LRF). These local register files store kernel constants, parameters, and local variables, reducing the required SRF bandwidth. Each cluster has 17 16-word LRFs for a total of 272 words per cluster and 2176 words across the eight clusters. Each local register file has one read port and one write port. The 17 local register files collectively provide 54.4 GB/s of peak data bandwidth per cluster, for a total bandwidth of 435.2 GB/s within the cluster array.

In order to provide the equivalent bandwidth and storage needed within a cluster using a single register file, a

[4] The memory system architecture could also support Direct RDRAM.

29-port, 256-entry register file would be required. This is significantly less efficient in terms of area and speed than the LRF organization. A simple storage cell for such a register file could be no less than 29 wiring tracks in both dimensions - one data wire and one control wire for each port. A more realistic storage cell would likely be much larger. An actual LRF storage cell is 6 wiring tracks by 8 wiring tracks. Therefore, the large multi-ported register file would require more than a 16x area increase over the 17 LRFs, making it the same size as three entire arithmetic clusters on Imagine.

The LRF organization is also faster than a single multi-ported register file for two reasons. First, LRFs hold 16 words instead of 256 words. This leads to much less capacitive loading and faster register file accesses by at least a factor of two. Second, with a large multi-ported register file, read and write accesses each require one wire delay across a cluster. In modern VLSI, this global communication incurs a significant delay for every register file access. With the LRF structure, this wire delay is only incurred for writes, since the reads occur locally to each ALU.

Additional storage is provided by a 128-word scratch-pad register file, the second unit from the right in Figure 5. It can be indexed with a base address specified in the instruction word and an offset specified in a local register. The scratch-pad allows for coefficient storage, short arrays, small lookup tables, and some local register spilling.

The intercluster communication unit, labelled `CU` in the figure, allows data to be transferred among clusters over the intercluster network using arbitrary communication patterns. The communication units are useful for kernels such as the Fast Fourier Transform, where interaction is required between adjacent stream elements.

The adders and multipliers are fully pipelined and perform single precision floating point arithmetic, 32-bit integer arithmetic, and 16-bit or 8-bit parallel subword integer operations, as found in MMX [9] and other multimedia extensions [8] [10]. The adders are also able to perform 32-bit integer and parallel subword integer shift operations. All multiplication and floating point addition has a latency of four cycles, while logical operations and integer addition have latencies of one and two cycles respectively.[5] The divide/square root unit is not pipelined and operates only on single precision floating point and 32-bit integers. The divider has a latency of 14 cycles for floating point divide, 13 cycles for floating point square root, and 21 cycles for integer divide.[5] This gives a total of up to 21 arithmetic operations in flight for each cluster (168 for all eight clusters), half of which are utilized by key media processing kernels. The 48 total arithmetic units, six units replicated across eight clusters, provide a peak computation rate of 16.2GOPS for both single precision floating point and 32-bit integer arithmetic. The rate for byte operations is 64.2GOPS (the divider does not perform subword operations).

3.4 Network Interface

The network interface connects the SRF to four bidirectional links (400MB/s per link) that can be configured in an arbitrary topology to interconnect Imagine processors. A send instruction executed on the source Imagine processor reads a stream from the SRF and directs it onto one of the links and through the network as specified by a routing header. At the destination Imagine processor, a receive instruction directs the arriving stream into the SRF. The send and receive instructions both specify *channels* to allow a single node to discriminate between arriving messages.

Using the stream model, it is easy to partition an application over multiple Imagine processors using the network. In the triangle rendering application of Figure 1, for example, higher throughput could be achieved by running the first three kernels on one Imagine, transmitting the output stream over the network to a second Imagine, and running the last five kernels on the second processor. The application is adapted by dividing the application-level code across the two processors, inserting a send instruction at one end, and inserting a receive instruction at the other.

3.5 Host Interface

A host processor issues application-level instructions to Imagine with encoded dependency information. The host interface buffers these instructions in an instruction window and issues them when their resource requirements and dependency constraints are satisfied. The host interface allows an Imagine processor to be mapped into the host processor's address space, so the host processor can read and write Imagine memory and can execute programs that issue the appropriate application-level instructions to the Imagine processor.

4. Programming Model

Figure 6 shows the actual code for the triangle rendering application, as presented in Figure 1. This C++ program executes on the host processor and issues application-level instructions that direct Imagine to perform stream operations. A set of library functions provide an interface to the application-level instructions. The `LOAD_MICROCODE` function loads the requested routine (e.g., `TRANSFORM.UC`) if it is not already in the control store, and returns its starting

[5] These latencies are derived from HSPICE simulations of the arithmetic units, including wiring parasitics extracted from the arithmetic unit layout.

```
void render_triangle_streams() {
  // Make sure the kernels are loaded into the Imagine microcontroller
  int transform    = load_microcode(“transform.uc”);
  int shade        = load_microcode(“shade.uc”);
  int project_cull = load_microcode(“project_cull.uc”);
  int span_setup   = load_microcode(“span_setup.uc”);
  int process_span = load_microcode(“process_span.uc”);
  int sort         = load_microcode(“sort.uc”);
  int compact      = load_microcode(“compact.uc”);
  int z_composite  = load_microcode(“z_composite.uc”);

  // Render a series of triangle streams
  for (int i=0; i<NUM_TRIANGLE_STREAMS; i++) {
    update_descriptors();
    stream_load(mem_model_triangles, srf_model_triangles);
    stream_op(transform,    srf_model_triangles,  srf_world_triangles);
    stream_op(shade,        srf_world_triangles,  srf_shaded_triangles);
    stream_op(project_cull, srf_shaded_triangles, srf_screen_triangles);
    stream_op(span_setup,   srf_screen_triangles, srf_spans);
    stream_op(process_span, srf_spans,            srf_fragments);
    stream_op(sort,         srf_fragments,        srf_sorted_fragments);
    stream_op(compact,      srf_sorted_fragments, srf_buf_idx,          srf_pixels);
    stream_load(mem_buf_pixels[srf_buf_idx], srf_pixels2);
    stream_op(z_composite,  srf_pixels,           srf_pixels2,          srf_output_pixels);
    stream_store(srf_output_pixels, mem_buf_pixels[srf_buf_idx]);
  }
}
```

Figure 6: Stream-Based Triangle Rendering

address. Memory load and store instructions are issued to Imagine by the STREAM_LOAD and STREAM_STORE functions, respectively. Finally, an operate instruction is issued by the STREAM_OP function causing the corresponding microcode kernel to run on each element of the specified source streams. For example, the first STREAM_OP function shown in the code initiates a compound stream operation on Imagine by issuing an operate instruction, as described in Section 3, specifying the start address of the TRANSFORM microcode. The instruction also specifies one input stream, SRF_MODEL_TRIANGLES, and one output stream, SRF_WORLD_TRIANGLES.

The arguments of the stream load, store, and operate instructions are specified by stream descriptors. Each memory stream descriptor (e.g., MEM_MODEL_TRIANGLES) includes a base address, length, record size, addressing mode, and stride or index stream. Each SRF stream descriptor (e.g., SRF_MODEL_TRIANGLES) includes a base location in the SRF, record length, and stream length. These descriptors are computed by C++ code running on the host processor (in the call to UPDATE_DESCRIPTORS in the figure).

Note that the microcode kernels need only be loaded once for all of the triangle streams that are to be processed, as the microcode store is large enough to hold all of the kernels. If another function were to be performed, then it is likely that these kernels would have to be evicted from the control store and reloaded the next time that RENDER_TRIANGLE_STREAMS is called. This allows the overhead of loading microcode kernels to be amortized across several large data streams.

Kernels, such as those invoked by STREAM_OP commands in Figure 6, are written in Imagine’s microassembly language which uses a C-like expression syntax. The microassembly code for the first kernel of the RENDER_TRIANGLE_STREAMS application, TRANSFORM, was shown in Figure 2. The kernel compiler takes this code as input and generates VLIW microcode instructions that will control the arithmetic clusters. The kernel compiler applies several common high level optimizations: loop unrolling, iterative copy propagation, and dead code elimination. It then performs list scheduling, starting with the largest, most deeply nested basic block. Within each block, the operations with the least slack are scheduled first.

Loops are the only control flow operations in the microassembly language; however, data dependent conditionals can be handled using the *select* instruction, scratch-pad accesses, or conditional streams. The *select* operation acts exactly like the C “?:” operator, and is implemented as a hardware primitive. The scratch-pad can be used to allow each of the eight clusters to access different data items based on a register offset computed in the clusters. Conditional streams are a powerful conditional mechanism allowing each cluster to independently conditionally read or write its streams.

Figure 7 graphically depicts the schedule output by the kernel compiler for the transformation kernel using software pipelining. The two loops of the kernel are compressed into one for efficiency, and the figure shows the 14 instruction software pipelined loop body. The 25 instructions before the loop that load constants and prime the software pipeline are not shown for clarity. The omitted setup code would only execute once for an entire stream, after which the loop would repeat for each set of eight vertices (one per cluster).

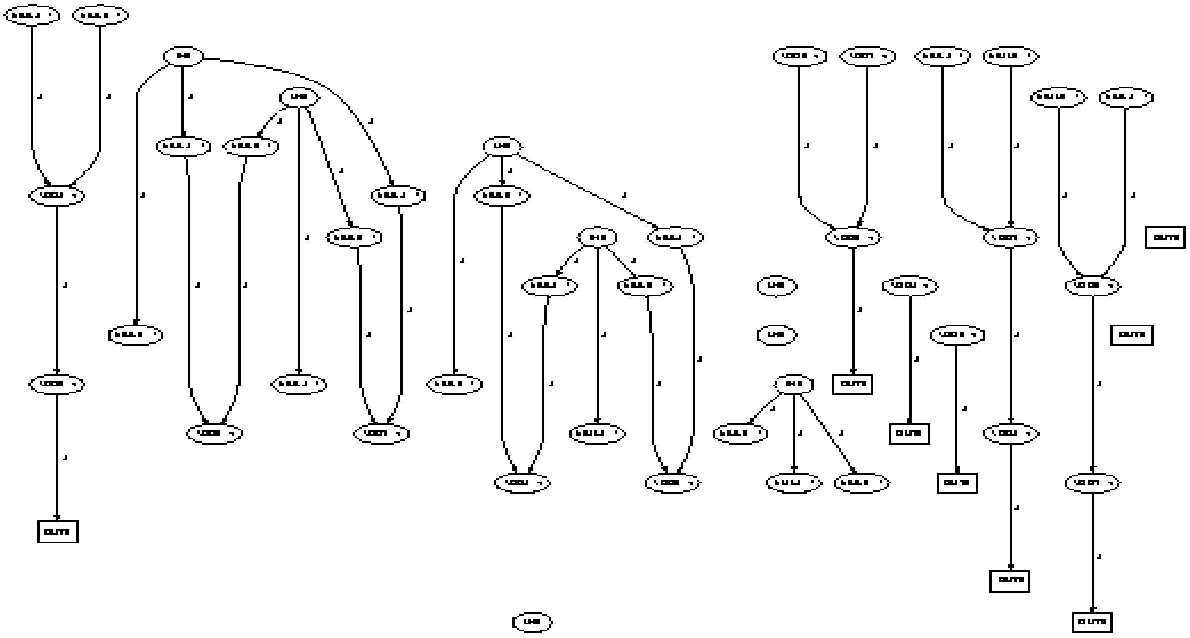

Figure 7: Compiled Transform Kernel

The nodes in the figure represent the operations in each VLIW instruction word of the compiled kernel. The arcs between the operation nodes indicate dependencies and are labelled with the latency of the source operation. Operation nodes which are not the source of a dependency arc interact only with operations on subsequent iterations of the software pipeline. Nodes that are aligned horizontally belong to the same instruction and specify operations that are issued simultaneously. For example, the first instruction of the loop specifies that each cluster will initiate two multiplications, and the second instruction will issue two multiplies, two adds, and a stream input operation to each cluster. The actual VLIW instruction word contains fields for all of the units, but nodes are only shown for operations that are not NOPs. For this particular kernel, 75% of the multiplier issue slots and 36% of the adder issue slots are utilized. The multipliers are the limiting resource that prevents the schedule from being compacted further.

5. Discussion

5.1 Storage Bandwidth Hierarchy

The bandwidth hierarchy of the Imagine architecture matches the demands of media applications to the capability of modern VLSI technology. The memory, SRF, and LRFs on a single Imagine chip have bandwidth ratios of 1:32:272. That is, for each word accessed in memory, 32 words may be accessed in the SRF, and 272 words may be accessed in the clusters' local register files. The memory bandwidth is set by the pin bandwidth limitations of the chip, and the SRF bandwidth is limited by the global wiring available on the chip. LRF bandwidth is less constrained because it involves no long wires, and is set by the number of ALUs. Viewed a different way, Imagine can perform 40.5 arithmetic operations per 4-byte word of memory bandwidth and 1.2 arithmetic operations per word of SRF bandwidth. In contrast, a conventional architecture requires 3 words of global register bandwidth per arithmetic operation. The bandwidth hierarchy enables Imagine to productively employ large numbers of ALUs without requiring prohibitive global bandwidth.

Imagine's bandwidth hierarchy is also well matched to the needs of media applications. Consider, for example, the triangle rendering application presented in Section 2. As mentioned earlier, for an image with a depth complexity of five, where an average triangle covers 25 pixels, rendering each triangle requires 1929 arithmetic operations, 666 SRF references, and 44 memory references (or 2.9 operations per SRF reference and 43.8 operations per memory reference).

5.2 Instruction Bandwidth

At peak, Imagine is able to execute 136 instructions[6] per cycle, not counting memory loads and stores. To achieve this rate, Imagine relies on an instruction bandwidth hierarchy with four levels: host interface, memory, control store,

[6] This includes 6 arithmetic instructions, 8 SRF references, one communication, and 2 scratch-pad references for each of the eight clusters.

and arithmetic units. A pair of application-level instructions issued through the host interface will transfer a microprogram from memory to the SRF and then from the SRF into the control store. This memory transfer will occur once for an entire stream operation, and since the working set of kernels for a typical application will fit in the control store, the kernel will be reused for several stream operations over the course of the application. Using strip-mining, a single application-level OPERATE instruction will iterate the kernel over a stream containing as many input records as will fit in the SRF. Each instruction in the inner loop of the kernel will be issued once for every eight input records, and will execute on all eight clusters simultaneously to process those eight records.

The triangle transform kernel can be used to illustrate these concepts. The kernel itself consists of 39 wide instructions, of which 14 are in the inner loop. Since each wide instruction can be specified with 12 32-bit words, 468 words will be transferred from memory to the control store for this kernel. This operation has a negligible impact on performance, as each individual scene can easily consist of hundreds of thousands of triangles, and other kernels can execute during the transfer. Each time the transform kernel runs, it will operate on 200 triangles.[7] Hence a single OPERATE instruction issued by the host processor will issue 1075 instructions from the control store and execute 8600 instructions on the eight clusters, each specifying up to 17 actual operations. So, even if the transform kernel is only run on one input stream, each instruction read from memory executes an average of over 200 times. As with the data bandwidth hierarchy, the instruction bandwidth hierarchy is well matched to the capabilities of current technology.

5.3 Conventional Processors

The advantages of Imagine's data and instruction bandwidth hierarchies can be appreciated by comparing them to conventional scalar and vector processors. Table 1 compares the memory, global register, and local register bandwidth requirements of a stream architecture (Imagine) with a vector processor and a scalar processor for the triangle transformation kernel.

Table 1. 32-bit Data References per Triangle for Transform

	Stream	Scalar		Vector	
Memory	5.5	117	(21.3)	48	(8.7)
Global RF	48	624	(13.0)	261	(5.4)
Local RF	372	N/A		N/A	

The left-most column of Table 1 shows the number of memory references, SRF references, and LRF references for the stream architecture. As stated in Section 2, for typical data the stream architecture performs 44 memory references for the entire pipeline. Amortizing this across the eight kernels gives 5.5 memory references per kernel. The transformation kernel itself reads 24 words from the SRF and writes 24 words to the SRF for a total of 48 SRF references. Finally carrying out the kernel requires 108 arithmetic operations per triangle that use 372 words of local register file (LRF) bandwidth. The additional LRF accesses beyond the 324 that would strictly be required by the 108 arithmetic operations come from register transfers required for software pipelining, and the register accesses to originally store data from the SRF into a local register file.

The next two columns of Table 1 compare these numbers to a scalar processor by giving both the absolute number of references and the ratio to a stream processor (in parentheses). The scalar numbers were generated by compiling the transformation kernel for an UltraSPARC II using version 2.7.2 of the gcc compiler with the -O2 flag. The table shows that a scalar architecture places very high demands on global register bandwidth, 13 times that of a stream processor. Scalar register files also increase the demand on memory bandwidth. Significant memory bandwidth is needed to load and store values from main memory because global scalar register files are too small to hold the local variables and parameters of a typical media kernel. In addition to the 48 words needed to fetch the input triangle and write the output triangle, 69 additional memory references are needed to load constants and input data repeatedly and to spill registers to the stack. As a result, the scalar processor requires 21.3 times as much memory bandwidth as a stream processor.

The last two columns present the number of references for the same kernel on a vector processor with an organization similar to the Imagine processor. All vector operations, however, are primitive arithmetic operations instead of compound stream operations. Also, data is accessed directly out of the global register file as vectors (streams with a record length of 1), instead of using local register files as a source and sink of data for the arithmetic units. A scalar register file to hold constants is assumed, but these references are not included as they are negligible compared to the vector accesses. Again the ratio of the number of references to a stream processor is given in parentheses. The global vector register file is a data bottleneck, as in the scalar case, since all computations must reference these global registers. Also, vectors of intermediate results must be recirculated through memory, as there is not enough temporary storage in a typical vector register file.

Vector processors reduce instruction bandwidth in a manner similar to Imagine because each instruction oper-

[7] This number is set by the number of 96-byte triangles that fit in the 64KB SRF.

ates on an entire vector, rather than a single word. Also, the vector register file relieves register pressure for scalars thereby reducing the number of main memory accesses that must occur during each record computation of a kernel. Even with these advantages, the vector processor still requires 8.7 times the memory bandwidth and 5.4 times the global register file bandwidth as a stream processor.

5.4 Cache Memories

A cache memory reduces main memory traffic in the same manner as a stream register file. However, a cache memory falls far short of a stream register file in terms of data and instruction bandwidth. The Imagine SRF is able to read 32 words per cycle because they are read sequentially from a single stream. In contrast, a typical cache memory allows only one or two independent word accesses per cycle, and thus is a bandwidth bottleneck for streaming media applications. Cache memories also require much greater instruction bandwidth than a SRF. An entire 2KW stream can be read from an SRF with a single application-level instruction. Reading the same data from a cache requires 2K instructions.

5.5 Sustained Kernel Performance

The Imagine kernel compiler and cycle accurate simulator were used to generate the following performance results for four representative media processing kernels: FFT, DCT, transform, and blockwarp. FFT performs one stage of an N-point complex, floating point, radix-2 Fast Fourier Transform. Streams must be run through the kernel $\log_2(N)$ times in order to perform the full transform. DCT performs the Discrete Cosine Transform on streams of 8×8 blocks of packed 16-bit fixed point data. Transform is the vertex transformation code shown in Figure 2, which operates on floating point pixel values. Blockwarp, taken from an image-based rendering application [5], performs a 3-D perspective transformation from model space into screen space on 8×8 blocks of 3-D floating point pixels. These four kernels are representative of typical graphics and signal processing applications. All performance numbers are sustainable for any data stream length that entirely fits within the SRF.

Table 2. Exploiting Data Parallelism

Kernel	1 to 8 Cluster Speedup
FFT	6.3
DCT	7.7
Transform	8.0
Blockwarp	7.8
Harmonic Mean	7.4

Table 2 shows the speedup of the four kernels on the Imagine processor going from a single cluster configuration to an eight cluster configuration. The near-linear speedup of 7.4 shows that the Imagine architecture is effective at exploiting the data parallelism available in these kernels. Vertex transformations are completely independent calculations, allowing the eight clusters to achieve a speedup of exactly eight for the transform kernel. Communication between the clusters reduces the speedup to less than 8 for the rest of the kernels, as they are not perfectly data parallel.

Table 3. Kernel Performance

Kernel	SRF BW (GB/s)	LRF BW (GB/s)	Arithmetic Ops Issued per Cycle	Arithmetic BW (GOPS)
FFT	24.24	186.70	21.21	8.48
DCT	4.79	134.10	41.12	16.45
Transform	14.63	113.37	20.57	8.23
Blockwarp	4.70	92.84	11.49	4.60
Harmonic Mean	7.53	123.44	19.31	7.73

Table 3 shows the sustained bandwidth used by these kernels and demonstrates the effectiveness of the bandwidth hierarchy. The kernels sustain an average of 7.7GOPS and require 123GB/s of local register bandwidth. The SRF, which provides higher bandwidth than a general-purpose global register file, cannot even provide half of the data bandwidth used by the arithmetic units. Therefore, without the small, fast local register files at the bottom of the bandwidth hierarchy, Imagine would not be able to achieve such high sustained performance on media kernels.

The arithmetic throughput of blockwarp is worse than the other kernels because it contains a divide in each pixel warp computation. The non-pipelined divider creates a bottleneck, even with software pipelining to hide its latency, because all subsequent calculations are dependent on its result. DCT makes use of 16-bit parallel subword operations in order to achieve a sustained throughput of 16.45GOPS.

Imagine’s stream architecture allows it to achieve high sustained performance for these media processing kernels. For the FFT kernel, an average of over 21 arithmetic operations are issued on every cycle for a sustained performance of 8.48GFLOPS. This represents a radix-2 butterfly computation every 2/3 cycles. The inherent parallelism in media applications and the Imagine bandwidth hierarchy results in similar sustainable performance on a variety of media processing applications.

6. Conclusions

The stream programming model exposes the parallelism and locality of media applications in a manner that is well matched to the capabilities of modern VLSI technology. A programmer describes an application as streams of records that are passed through computation kernels. Individual stream elements may be operated on in parallel to exploit data parallelism. Instruction-level parallelism can be exploited within the individual computation kernels. Also, control parallelism can be exploited by partitioning an application across multiple processors. Locality is exposed both by recirculating streams through a stream register file and also within the computation kernels which access streams in order and keep a small set of local variables. For triangle transformation, a typical media kernel, this locality reduces the demand on global register and memory bandwidth over a scalar processor by factors of 13 and 21, respectively. This enables a stream architecture to make efficient use of a large number of arithmetic units without global bandwidth becoming a bottleneck.

Imagine is a single-chip media processor that supports the stream programming model by providing a data bandwidth hierarchy matched to the demands of typical media applications. Imagine is organized around a central stream register file (SRF) and all operations on Imagine are performed by transferring streams to and from the SRF. Load and store instructions transfer streams between the SRF and memory. Network instructions transfer streams between the SRFs on different Imagine processors. Stream computations are performed by passing a stream from the SRF through an array of 48 32-bit FP arithmetic units (6 units in each of 8 clusters) and back to the SRF. These units have a peak performance of 16.2GFLOPS (32-bit) and achieve 8.5GFLOPS on important media kernels.

At present (Summer 1998) we have developed a cycle accurate simulator of Imagine. Designs of key components, including the 32-bit segmented adder and multiplier and key register files have been carried through sized schematics and layout to accurately validate area and delay estimates. A programming system including a kernel compiler and application toolbench has been developed. The kernel compiler accepts a kernel in a restricted subset of C and a description of an arithmetic cluster and outputs optimized microcode. The application toolbench provides utilities that allow a host processor to issue application-level instructions to Imagine.

Our experiments to date have demonstrated the ability of stream architectures to reduce demands on memory and global register bandwidth and thus achieve high sustained throughput rates on key kernels. There is much work to be done, however. Trade-offs in the composition of an arithmetic cluster must be quantified as do the trade-offs between large clusters and more clusters and between a single stream processor with many clusters and many stream processors with a few clusters. Conditional operations also present an interesting challenge in a machine that has 168 operations in flight each cycle. Many memory system issues are also open including how to partition addresses across the banks, how to order memory accesses, how to exploit a limited amount of cache memory, and how to use large amounts of on-chip DRAM, if available. To work seamlessly with a host processor, the host interface presents challenges such as how to sequence instructions, how to synchronize with the host, and how to pass data between the host and Imagine.

7. Acknowledgments

We would like to thank Brad Johanson for his tireless effort at application development for Imagine. We would also like to thank Chris Buehler, J.P. Grossman, and Don Alpert for their contributions to the Imagine architecture.

References

[1] BOVE, JR. V.M. AND WATLINGTON, J.A. Cheops: A Reconfigurable Data-Flow System for Video Processing. *IEEE Transactions on Circuits and Systems for Video Technology* (April 5, 1995), pp. 140-149.

[2] CONTE, THOMAS M., ET. AL. Challenges to Combining General-Purpose and Multimedia Processors. In *Computer* (December 1997), pp. 33-37.

[3] DIEFENDORFF, K. AND DUBEY, P. How Multimedia Workloads Will Change Processor Design. In *Computer* (Septemeber 1997), pp. 43-45.

[4] FOLEY, JAMES D., ET. AL. *Computer Graphics: Principles and Practice*, Addison-Wesley Publishing Company: Menlo Park, California, 1996.

[5] GROSSMAN, J.P. AND DALLY, WILLIAM J. Point Sample Rendering. In *Proceedings of the 9th Eurographics Workshop on Rendering* (June, 1998), pp. 181-192.

[6] KIRKPATRICK, SCOTT, Presentation at Harvard University, 1996.

[7] LIAO, HENG AND WOLFE, ANDREW. Available Parallelism in Video Applications. In *Proceedings of the International Symposium on Microarchitecture* (December, 1997), pp. 321-329.

[8] LEE, RUBY B. Subword Parallelism with MAX-2. In *IEEE Micro* (August, 1996), pp. 51-59.

[9] PELEG, ALEX AND WEISER, URI. MMX Technology Extension to the Intel Architecture. In *IEEE Micro* (August, 1996), pp. 42-50.

[10] TREMBLAY, MARC, ET. AL. VIS Speeds New Media Processing. In *IEEE Micro* (August, 1996), pp. 10-20.