Appears in the Proceedings of the 2002 International Conference on Computer Design.

A Stream Processor Development Platform

Ben Serebrin[†], John D. Owens[†], Chen H. Chen[⋆], Stephen P. Crago[⋆], Ujval J. Kapasi[†]
Brucek Khailany[†], Peter Mattson[†], Jinyung Namkoong[†], Scott Rixner[‡], William J. Dally[†]

[†] *Computer Systems Laboratory*
Stanford University
Stanford, CA 94305
{*serebrin, jowens, kapasi, khailany, pmattson, namkoong, billd*}*@cva.stanford.edu*

[‡] *Computer Systems Laboratory*
Rice University
Houston, TX 77005
rixner@rice.edu

[⋆] *Information Sciences Institute East*
University of Southern California
Arlington, VA 22203
{*cchen,crago*}*@east.isi.edu*

Abstract

We describe a hardware and software platform for developing streaming applications. Programmers write stream programs in high-level languages, and a set of software tools maps these programs to code that runs on a streaming hardware system. The hardware platform includes two Imagine Stream Processors, together providing 32 GFLOPS peak performance, and a high-speed onboard network to carry video and other data between peripherals and the Imagine processors.

1 Introduction

In this paper, we describe a development platform for the Imagine Stream Processor and show how high performance streaming media applications are realized on this platform. The Imagine development system supports two Imagine processors and provides them with memory, video inputs and outputs, and a high-speed network, all on a single circuit board that allows us to test and debug the Imagine hardware and software systems and to develop Imagine applications. A single Imagine node can provide 16 GFLOPS peak performance. Imagine's network interface allows multiple processors and media devices to be connected in a scalable network. The development board provides sufficient computational power for real-time video applications such as stereo depth extraction and MPEG encoding.

Imagine applications are programmed using StreamC and KernelC. StreamC programs specify data movement and initiate kernel operations on entire streams of data. A host processor executes these stream programs and uses a run-time scheduler to coordinate stream traffic. KernelC kernels, which execute on the Imagine processor, perform computation on the data records within a stream.

This paper describes the hardware and software systems we have implemented to operate a dual-Imagine system. In Section 2, we describe an application we will use to illustrate the details of the system throughout the paper. Section 3 discusses the programming environment and stream and kernel schedulers. In Section 4 we detail the hardware implementation of the system. Finally, we describe our plans to extend the design of the dual-Imagine system to a 64-Imagine computer capable of over 1 TeraFLOPS in Section 5.

2. Example Application: 3-D Teleconferencing

In this section, we sketch how a 3-D teleconferencing application may be mapped to the Imagine development system. The approach taken for this implementation applies to any Imagine application. We will refer to this example as details of the development platform are introduced later in this paper. Teleconferencing is particularly suited to this system because it exercises all of the system's components, including video I/O, the Imagine network, the host processor, and the Imagine itself. Video applications such as this one perform well in the Imagine stream processing architecture because pixel data can be streamed in and passed through several computation kernels within the stream processor, taking advantage of producer-consumer locality before results are finally streamed out of the processor. Applications similar to teleconferencing will be used in our evaluation of the Imagine architecture.

A 3-D teleconferencing setup requires two or more cameras that operate in concert to allow extraction of depth information. The cameras are located a small, fixed distance from each other and aimed at the same scene to allow binocular depth extraction from the resulting overlapping images.

The Imagine stream processors perform stereo depth extraction on two video input streams [1] and output a stream of depth-annotated, MPEG compressed image data. The output stream is transferred to the remote viewing site, where MPEG decoding and 3-D image reconstruction must be performed.

The major computational kernels for the transmission side of the application have been written and optimized for Imagine. The following results are derived from performance numbers from our cycle-accurate simulator for the depth extraction and MPEG kernels. These computationally intense tasks fit well on a single Imagine chip, which can process 42 frames per second for 640x480 stereo depth extraction and 94 fps for MPEG encoding (derived from Khailany et al. [3], using a core frequency of 400 MHz). Using these two kernels to create a depth map from two images and to compress a single composite image, an Imagine could sustain approximately 29 fps for the transmission side of 3-D teleconferencing.

Data originates in this application with two digital cameras, which are connected to the board via Firewire ports. Frames from both cameras travel over the Imagine network on the board into the stream processor, where the images are processed and then sent out via an external network to the viewing station for the teleconference. While one Imagine has sufficient resources to execute both depth-extraction and MPEG encoding, the application may be partitioned between the processors to increase performance, reduce latency, or to further process the image. We will describe the use of multiple Imagines in more detail in section 5.

3. Software Implementation

The Imagine stream programming model decomposes an application into a series of computation kernels that operate on streams of data. Streams are sets of sequential data records that lend themselves to high-performance computation through their regular and predictable structure. Khailany et al. show how the depth extractor portion of the teleconferencing application maps to the stream programming model [3].

Stream and Kernel programs are written in the *StreamC* and *KernelC* languages, respectively. The use of these languages for Imagine programming allows the programmer to use the familiar high-level C language instead of writing in Imagine assembly code. The two-level programming system reflects the division of labor on the Imagine system: the kernels running on Imagine perform numeric computation, while the stream programs running on the Host Processor correspond to a high-level description of the application.

Several types of programs work together to execute an Imagine application, as illustrated in figure 1. Each *kernel program* takes one or more data streams as inputs and produces one or more streams as outputs. For example, one kernel in the MPEG-2 compression portion of the application is Discrete Cosine Transform (DCT), which accepts a stream of 8x8 pixel blocks, performs a 2-D DCT on them, and outputs a stream of transformed blocks. *Stream programs* run on the Host Processor, call the kernel programs, and initiate the transfer of data streams. The Host Processor's runtime libraries interface the execution of Imagine applications with the console PC, provide a command-line interface, and load compiled StreamC and KernelC code into the Host Processor and Imagine. Finally, the Console PC runs a user interface application and a simple PCI device driver. The software exports the Console PC's console I/O and file system to the Host Processor, and may be extended for specific applications to provide network functions and a more elaborate user interface.

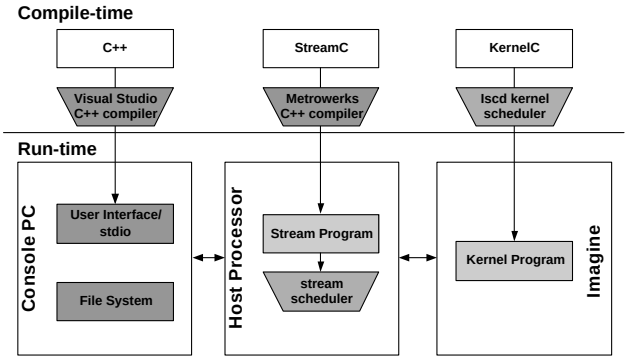

Figure 1. Imagine System Software Layout

The way kernels work together in a real application can be seen by revisiting our 3-D teleconferencing application. First, the image-sharpening kernel enhances the contrast for a stream of pixels, then the depth-extraction kernel performs sum-of-absolute-differences calculations to output a stream of pixels that represents a depth map. Finally, the MPEG compression kernels read image pixels in as 8x8 blocks and output a stream of compressed blocks. The overall teleconferencing application is implemented as a StreamC program that instructs Imagine to receive network data, run kernels, and then send out the resulting data. The Console PC is used to provide a command line interface to allow the user to start up and control the application.

3.1. StreamC and the Stream Scheduler

StreamC provides a simple and complete programming interface for stream programs. StreamC programs are written in C++ with calls into a library of stream functions, which enables kernel calls and stream manipulation. Kernel calls are treated as functions that take input and output

streams as arguments. StreamC provides functions to create streams, transfer them over the Imagine network, and communicate with the Console PC.

The *stream scheduler* allocates space in external memory and in the *Stream Register File (SRF)*, a large on-chip memory used to store streams that are passed between kernels. The scheduler determines when to place streams in the SRF and when to move them to and from main memory to take advantage of producer-consumer locality between different kernels [2]. Efficient stream programs minimize traffic between the SRF and memory. *Stripmining* breaks a large input stream into smaller batches when the data is larger than the SRF. In our example, the entire data set of two video images is larger than the SRF, so the stream scheduler divides the images into batches of pixel rows. One batch is loaded into the SRF at a time. The entire series of kernels executes on this batch before computation begins on the next rows. In this way, raw pixel data arrives in the SRF and passes between kernels through temporary storage in the SRF. Expensive memory traffic is encountered only when final results are written out.

To ensure high resource utilization, the stream scheduler performs software pipelining at the stream level, where kernels are executed concurrently with memory operations. In our example, kernels process a batch of pixels while the scheduler simultaneously sends out finished data to memory and loads the next rows into the SRF. One measure of the efficacy of software pipelining is *occupancy*, the percentage of time that a module is performing work. In practice, software pipelining improves the occupancy of the most heavily used module (either the memory system or the arithmetic units) to above 90% [4].

3.2. KernelC and the Kernel Scheduler

Kernel programs are written in KernelC, a subset of C. KernelC programs operate on single stream elements and are repeatedly executed on each element of a stream. Because of this simplification, KernelC does not contain any control structures except loops and select statements and does not support subroutines. It supports Imagine's four basic datatypes: integers, paired halfwords in a single word, four bytes in a single word, and single-precision floating point. Records composed of these datatypes are also supported. KernelC also provides an interface to the various mechanisms on Imagine that implement data-dependent conditionals.

The kernel scheduler, IScd, compiles the KernelC description of the kernel into Imagine executable code. Imagine has several functional units operating in parallel, so the kernel scheduler must extract instruction-level parallelism from the kernel. The algorithm used in IScd, *communication scheduling*, is described by Mattson [5].

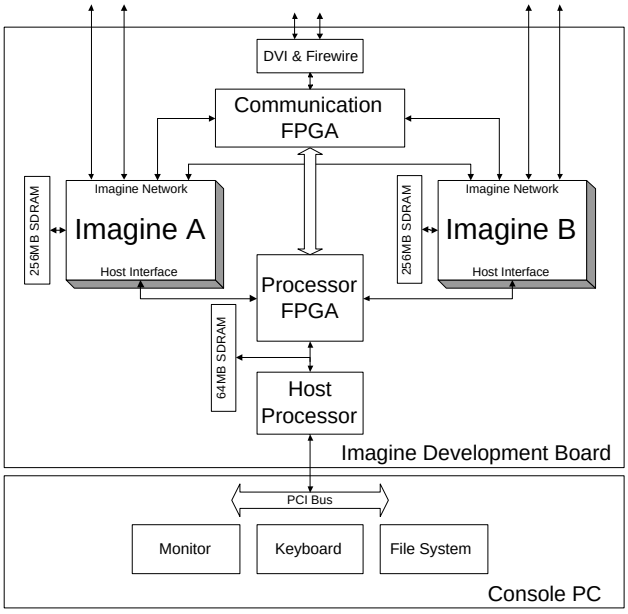

Figure 2. Imagine Prototype Board Architecture

3.3. Compilation and Booting

Three different types of programs are compiled and loaded on the Imagine system. Console PC programs are compiled in Microsoft Visual Studio and run as standard Microsoft Windows 2000 applications that access a device driver for communication over PCI to the Host Processor. Both StreamC programs and the stream scheduler are compiled for the Host Processor with Metrowerks Codewarrior for Embedded PowerPC. On boot, the Host Processor reads a small boot program from Boot ROM and waits for the Console PC to send the full StreamC application. Finally, IScd compiles KernelC code for Imagine. The stream scheduler loads compiled kernels into Imagine's microcode store.

4. Hardware Implementation

The Imagine development system contains two Imagine processors with a high-speed network, I/O devices, and a Host Processor. Two FPGAs serve as translators to make Imagines and I/O devices available to the Host Processor and on the Imagine network. The Host Processor controls the entire prototype system, running stream programs and a stream scheduler. Figure 2 shows the architecture of the Imagine system. A photograph of the prototype board without the Imagine processors is shown in Figure 3. The prototype board is a 14-layer, 31x27cm oversized PCI card with

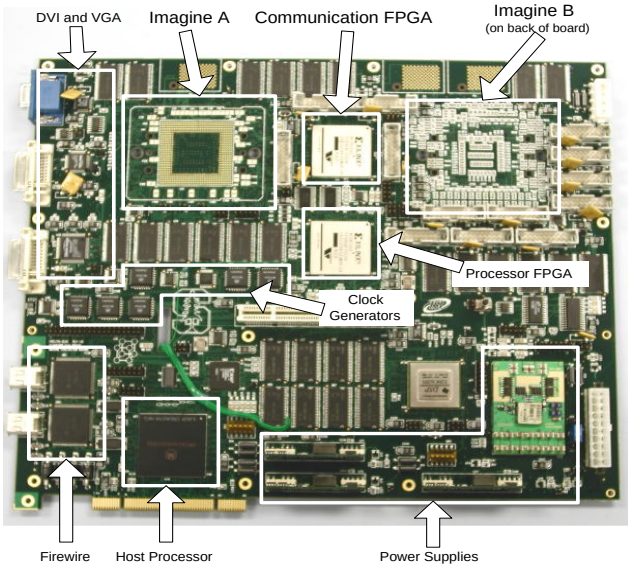

Figure 3. Imagine Prototype Board Photograph

substantial real estate dedicated to debugging tools such as probe points, configurable clocks, and adjustable power supplies.

4.1. Imagine

The Imagine Stream Processor receives data streams from the network and host interfaces and runs arithmetic kernels on this stream data. To support the streaming model, Imagine has a hierarchical register file organization. A large Stream Register File provides on-chip storage for transferring data streams between kernels [3]. The stream scheduler utilizes this on-chip capacity to keep external communication to a minimum, as described in Section 3.1. In addition to its internal memory, Imagine can address up to 512 MB of external SDRAM. The development system provides each Imagine with 256 MB. Imagine has a total of 48 floating-point ALUs organized as 8 SIMD clusters, providing 16 GFLOPS of peak performance. The Imagine Stream Processor is implemented in a 0.15 micron Texas Instruments process.

4.2. Network

The Imagine network provides high-speed communication for data streams. Each Imagine chip has 4 bidirectional, 16 bit wide, 400 MHz network ports that can be used to create scalable Imagine networks. Streams are composed of a series of 80-bit flits, of which 16 bits are used for control and 64 are data. Each stream starts with a head flit that contains routing information, a tag to identify the stream, and a virtual channel number. Subsequent body flits of the stream are identified by the virtual channel number. A stream may be of arbitrary length; each stream is ended by a tail flit. Each virtual channel uses credit-based flow control. All network transfers originate and terminate in the Imagine's Stream Register File or at a peripheral. Network streams are statically source-routed by the Stream Scheduler.

In our example application, the two cameras send data to Imagine via the network. We use tags to distinguish images from the left and right cameras. Because an image is larger than the Imagine's SRF, the Stream Scheduler stages the transfer of image data from the cameras through the SRF to memory. Network and memory transfers operate independently of all other Imagine operations. While a row is being transferred to memory, the next row of pixels is arriving and the arithmetic clusters are operating on another portion of the image.

A router on each Imagine chip enables networks to be constructed with no external logic. In a multiple-Imagine system, packets move through a series of Imagines on the routes specified in their headers. The network supports arbitrary topologies of node degree 4 or less, e.g., a 2-D torus. The network on the development board contains two Imagines and media devices as shown in figure 2. The off-board Imagine network connections allow the network to grow by connecting to additional dual-Imagine boards with a passive backplane. Because the aggregate bandwidth increases with the number of nodes in the Imagine network, the network can scale to support hundreds of Imagines and peripherals. In contrast, simple bus-based or shared-medium interconnects such as PCI and Ethernet have fixed bandwidth even as devices are added.

4.3. Media Peripherals and Device Memory Mapping

The prototype board supports several high-bandwidth media devices. A DVI (Digital Video Interface) input channel allows video input, and DVI and VGA output channels serve as the video output for Imagine's graphics applications. Two 400 Mbps Firewire channels allow high-resolution cameras or other devices to provide input for video applications.

The Imagine board contains two Xilinx VirtexII-1000 FPGAs for memory translation and peripheral control. The Processor FPGA maps the control and data registers for the Imagine chips and peripherals into the Host Processor's memory space. The Communication FPGA connects all the peripherals to the Imagine network and translates between their native interfaces and the Imagine network. In addition,

the Communication FPGA implements a double buffer using external memory that can be used as a frame buffer in video rendering applications. The chosen FPGAs can accommodate large designs, and thus may be programmed with additional functions for specific applications. General uses include staging incoming data for convenient stream representation and performing inherently serial pre- or post-processing on data sets, such as the run-length encoding stage of MPEG compression.

In the teleconferencing application, the Communication FPGA translates the incoming video images into network streams, tags them appropriately, and sends them to the Imagine processor. In a teleconference receiver, the reconstructed video output is output using DVI or VGA.

4.4. Host Interface and Host Processor

In addition to its four high-speed network interfaces, Imagine has a conventional TTL interface to connect to the Host Processor. Imagine is controlled through this interface; programs and data may be loaded and Imagine's control and status registers are accessible through the host interface. The host interface is connected to the Host Processor, an embedded PowerPC 8240, through the Processor FPGA. The 8240 was chosen because it provides a built-in PCI interface and reasonable performance in one chip.

5 Future Work

The Imagine prototype board is designed as an application development platform and to test all components of a larger system of 64 Imagine nodes. A unit of two Imagines and one PowerPC host processor, plus memory, can be duplicated four times on a board, and eight boards can be placed in a system, to create a 64-Imagine machine capable of 1 TeraFLOPS of computing. The Imagines will be tied together in a 2-dimensional, 8x9 torus network, where the 9th column of the torus is occupied by shared I/O devices such as video inputs and outputs, consoles, hard drives, and network ports. A PCI bus on each card ties the four host processors on that card together for program distribution and control, and bridge chips on each card connect the PCI busses together. Figure 4 shows the design of the 64-Imagine system. The system will have a passive backplane to connect eight 8-Imagine boards and an I/O board together in a unified, high-speed network of computation nodes, with a bridged PCI bus for host-processor coordination.

An application must be partitioned to run on a multiple Imagine system. Applications can be partitioned either by kernels, by data, or by both kernels and data. With kernel partitioning, the kernels of an application are divided across the processors in a system and network communications are used to pass streams between kernels running on different

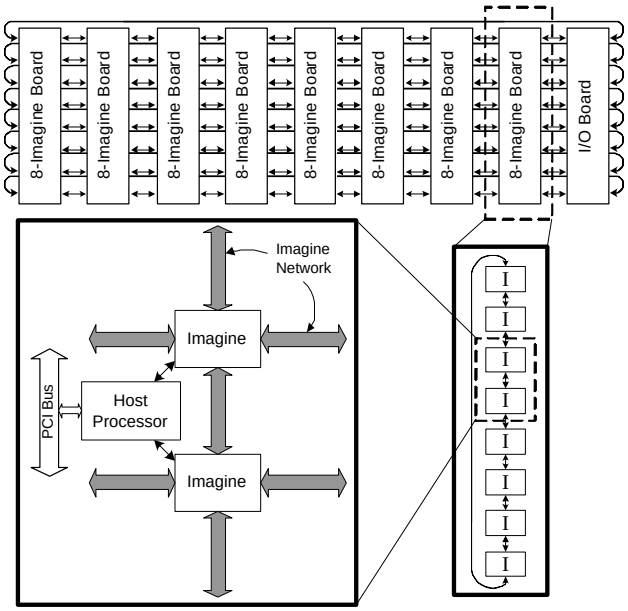

Figure 4. A TeraFLOPS system containing 64 Imagines

processors. Data partitioning is achieved by replicating a kernel on a number of processors and dividing the data between them - e.g., sending even records of a stream to the kernel running on node 1 and the odd elements of a stream to the identical kernel running on node 2. Manual and automated partitioning of streaming programs are currently under investigation.

6 Conclusion

The Imagine development platform has three functions: Imagine architecture evaluation, software development, and component debugging for the 64-Imagine TeraFLOPS system. The development platform, which is successfully running simple applications as of July 2002, will allow us to verify the performance of the Imagine design, which has been previously exercised with a suite of applications running on our cycle-accurate simulator. We will distribute the Imagine board to academic partners who will explore the uses of streaming in graphical and scientific computing applications, for both single and multiple-Imagine applications. The experience gained in implementing and programming the dual-Imagine system will guide us in the construction of a 1-TeraFLOPS streaming computer.

Acknowledgments

We would like to thank Jinwoo Suh and Li Wang of ISI East for their work in implementing and debugging board software and FPGA code. Thanks also go to Abhishek Das and Jung Ho Ahn at Stanford for their help in debugging system hardware and software. Finally, Alan Swithenbank provided invaluable advice and assistance with delicate circuit board rework.

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-C0037, by ARPA order L172 monitored by the Department of the Air Force under contract F29601-00-2-0085, by Intel Corporation, by Texas Instruments, by an Intel Foundation Fellowship, and by the Interconnect Focus Center Program for Gigascale Integration under DARPA Grant MDA972-99-1-0002.

References

[1] T. Kanade, H. Kano, S. Kimura, A. Yoshida, and K. Oda. Development of a video-rate stereo machine. In *Proceedings of the International Robotics and Systems Conference*, pages 95–100, Pittsburgh, PA, August 5–9, 1995.

[2] U. J. Kapasi, P. Mattson, W. J. Dally, J. D. Owens, and B. Towles. Stream Scheduling. Concurrent VLSI Architecture Tech Report 122, Stanford University, Computer Systems Laboratory, March 2002.

[3] B. Khailany, W. J. Dally, S. Rixner, U. J. Kapasi, P. Mattson, J. Namkoong, J. D. Owens, B. Towles, and A. Chang. Imagine: Media Processing with Streams. *IEEE Micro*, pages 35–46, Mar/Apr 2001.

[4] P. Mattson. *Programming System for the Imagine Media Processor*. PhD thesis, Stanford University, 2002.

[5] P. Mattson, W. J. Dally, S. Rixner, U. J. Kapasi, and J. D. Owens. Communication scheduling. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 82–92, 2000.