Evaluating the Imagine Stream Architecture

Jung Ho Ahn, William J. Dally, Brucek Khailany, Ujval J. Kapasi, and Abhishek Das
Computer Systems Laboratory
Stanford University, Stanford, CA 94305, USA
{gajh,billd,khailany,ujk,abhishek}@cva.stanford.edu

Abstract

This paper describes an experimental evaluation of the prototype Imagine stream processor. Imagine [8] is a stream processor that employs a two-level register hierarchy with 9.7 Kbytes of local register file capacity and 128 Kbytes of stream register file (SRF) capacity to capture producer-consumer locality in stream applications. Parallelism is exploited using an array of 48 floating-point arithmetic units organized as eight SIMD clusters with a 6-wide VLIW per cluster. We evaluate the performance of each aspect of the Imagine architecture using a set of synthetic micro-benchmarks, key media processing kernels, and full applications. These micro-benchmarks show that the prototype hardware can attain 7.96 GFLOPS or 25.4 GOPS of arithmetic performance, 12.7 Gbytes/s of SRF bandwidth, 1.58 Gbytes/s of memory system bandwidth, and accept up to 2 million stream processor instructions per second from a host processor.

On a set of media processing kernels, Imagine sustained an average of 43% of peak arithmetic performance. An evaluation of full applications provides a breakdown of where execution time is spent. Over full applications, Imagine achieves 39.4% of peak performance, of the remainder on average 36.4% of time is lost due to load imbalance between arithmetic units in the VLIW clusters and limited instruction-level parallelism within kernel inner loops, 10.6% is due to kernel startup and shutdown overhead because of short stream lengths, 7.6% is due to memory stalls, and the rest is due to insufficient host processor bandwidth. Further analysis included in the paper presents the impact of host instruction bandwidth on application performance, particularly on smaller datasets. In summary, the experimental measurements described in this paper demonstrate the high performance and efficiency of stream processing: operating at 200 MHz, Imagine sustains 4.81 GFLOPS on QR decomposition while dissipating 7.42 Watts.

1. Introduction

Media applications such as video processing, wireless communication, and 3-D graphics are pervasive and computationally demanding. To decompress, deinterleave, and scale an HDTV video stream in real time, for example, requires billions of operations per second. Fortunately these applications are characterized by ample parallelism. Most of these applications are served today by special-purpose ASIC processors containing hundreds to thousands of ALUs. While such ASIC solutions are efficient, they lack flexibility and are not feasible for certain low-volume applications.

Imagine [8] is a programmable stream processor aimed at media applications. Expressing an application as a stream program, sequences of records flowing through computation *kernels*, exposes both parallelism and locality. Imagine exploits the parallelism of a stream program with an array of 48 32-bit floating-point units. Two levels of register files, 9.7 KBytes of local register files and 128 KBytes of stream register file, capture the locality of stream programs, enabling a high ratio of arithmetic to off-chip bandwidth. By keeping most data transfers local (over 95% of all transfers are from local registers) Imagine offers efficiency approaching that of an ASIC while retaining the flexibility of a programmable processor.

This paper describes the experimental evaluation of a prototype Imagine processor fabricated in an 1.5 Volts, 0.18 μm CMOS process[1] and packaged in a 768-pin BGA package. A set of micro-benchmarks are used to measure the performance of each feature of Imagine independently. Media kernels and applications are then used to measure the overall performance of Imagine on actual workloads.

Imagine achieves 7.96 GFLOPS / 25.4 GOPS ALU performance on synthetic benchmarks and sustains between 16% and 60% of this peak performance on four media applications. On all of these applications, a single Imagine chip is able to sustain greater than real-time performance. On MPEG encoding, for example, Imagine sustains a compression rate of 138 frames per second while consuming 6.8 Watts. At the kernel level, the difference between peak and sustained performance is primarily due to idle ALUs caused by limited ILP and load imbalance across ALUs. On three

---

1   The process used to fabricate Imagine has 0.18 μm metal rules with 0.15 μm devices.
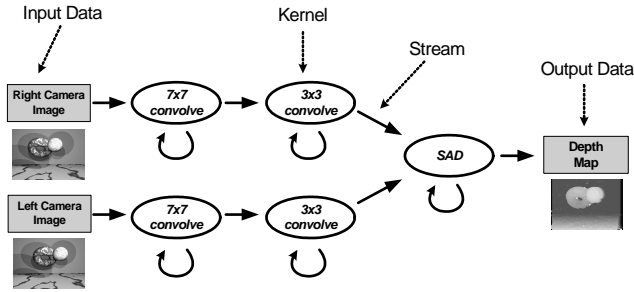
Figure 1. Stream and kernel diagram for stereo depth extractor application

of the four applications, less than 10% of performance is lost due to overhead at the application level — memory and host interface stalls.

The remainder of this paper describes the experimental evaluation of Imagine in detail. Section 2 describes our experimental setup including the Imagine processor and supporting hardware and software infrastructure. Results of the micro-benchmark programs, used to measure and verify the performance and power efficiency of Imagine components, are described in Section 3. Results from full applications, described in Section 4, show how these components interact. Section 5 highlights the effectiveness of stream processing and analyzes factors causing differences in the results between simulation and experimental measurement.

2. The Imagine Stream Processing System

The Imagine stream processing system is composed of a programming model, architecture, compiler tools, and a development board all organized to take advantage of the locality and parallelism inherent in media applications. This system allows stream processors to provide the performance of a special-purpose processor on media applications while retaining the flexibility of a programmable engine.

2.1. Stream Programming Model

Media processing applications have considerable producer-consumer locality, are data-parallel, and are compute-intensive [13]. The stream programming model exposes these characteristics by representing applications as a set of computation kernels that consume and produce data streams. Each data stream is a sequence of data records of the same type, and each kernel is a program that performs the same set of operation on each input stream element, and produces one or more output streams. Media applications are programmed using this model with two languages developed for Imagine: *StreamC* and *KernelC*.

A StreamC program specifies the order of kernel executions and organizes data into sequential streams that are passed from one kernel to the next. For example, Figure 1 shows a graphical representation of the StreamC program

for a stereo depth extractor application [5]. This application processes images from two cameras that are positioned at a horizontal offset from each other. The images are first pre-processed by two convolution kernels (3x3 convolve and 7x7 convolve). Then the Sum-of-Absolute-Differences (SAD) kernel is called repeatedly to find the number of pixels of horizontal shift that minimizes the SAD of a 7x7 area centered around each pixel. Each call to the kernel searches a different set of horizontal offsets, or disparities. The depth at each pixel is proportional to the inverse of the best disparity match. Notice, in the example above, that data in each output stream is destined either for the next kernel in the application pipeline, or is consumed by the next call to the same kernel. Furthermore, though not shown in the diagram, the programmer can embed arbitrary C/C++ in a StreamC program.

Kernels in the stream programming model are programmed in KernelC. They are structured as loops that process element(s) from each input stream and generate output(s) for each output stream during each loop iteration. KernelC disallows any external data accesses besides input stream reads, output stream writes, and a few scalar parameter inputs. Kernels are also written so that successive iterations of the main loop can be executed concurrently using parallel hardware.

Mapping media applications to StreamC and KernelC exposes the available parallelism and locality. Producer-consumer locality between subsequent kernels is exposed as the streams passing between kernels in StreamC. Locality within kernels is captured in KernelC. Data-level parallelism is exposed by both StreamC and KernelC because kernels perform the same computation on all of the elements of an input stream.

2.2. Imagine Architecture

The Imagine architecture directly exploits the parallelism and locality exposed by the stream programming model to achieve high performance. As shown in Figure 2, Imagine runs as a coprocessor to a host. The host processor executes scalar code compiled from a StreamC program and issues stream instructions to Imagine via an on-chip stream controller. The stream register file (SRF), a large on-chip storage for streams, is the nexus of Imagine. All stream instructions operate on data in the SRF. Stream load and store instructions transfer streams of data between the SRF and memory (possibly using indirect (gather/scatter) addressing). Kernel-execute instructions perform a kernel (such as 7x7 convolve or SAD from Figure 1) on input streams of data from the SRF, and generating output streams in the SRF.

The eight arithmetic clusters execute kernels in an eight-wide SIMD manner. Each arithmetic cluster, shown in Figure 3, contains six 32-bit floating-point units (FPUs) (three adders, two multipliers, and a divide square-root unit). Each FPU except for the divide square-root unit is fully pipelined to execute a single-precision floating-point, 32-bit integer,
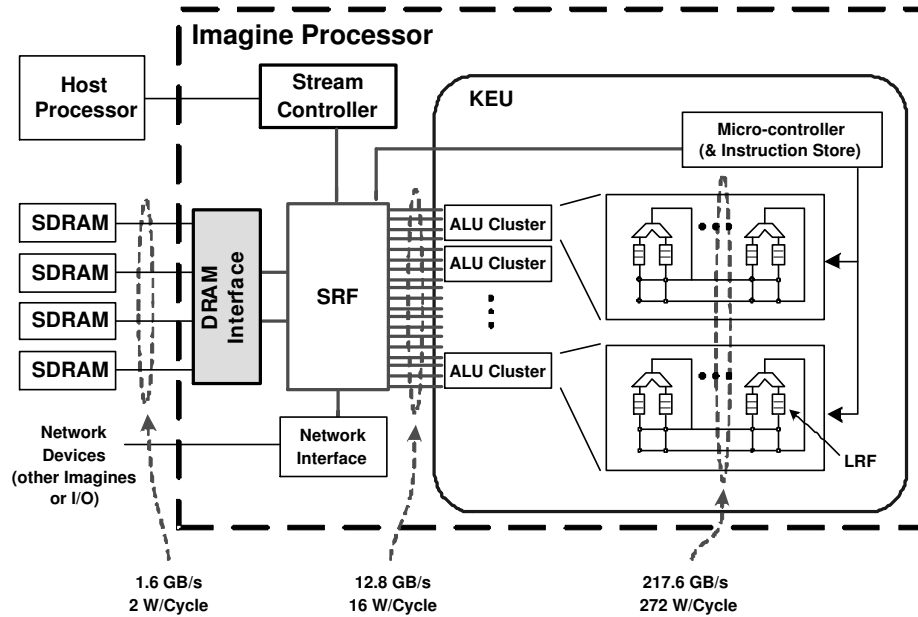
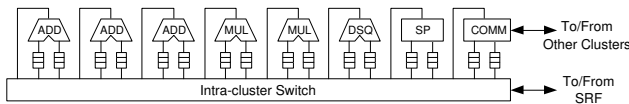**Figure 2. Diagram of Imagine architecture with bandwidth hierarchy numbers**

Figure 3. Imagine arithmetic cluster

or multiple lower-precision integer operations per cycle. These six FPUs execute kernel VLIW instructions issued from a single micro-controller each clock cycle. Two-port local register files (LRFs) feed the inputs of each FPU and an intra-cluster switch connects the outputs of the ALUs and external ports from the SRF to the inputs of the LRFs. In addition, a scratchpad (SP) unit is used for small indexed addressing operations within a cluster, and an intercluster communication (COMM) unit is used to exchange data between clusters.

When a kernel-execute instruction is received from the host processor, the micro-controller starts fetching and issuing VLIW instructions from the microcode instruction store. For each iteration of a typical kernel's inner loop, the eight clusters read eight subsequent elements in parallel from one or more input streams residing in the SRF, each cluster executes an identical series of VLIW instructions on stream elements, and the eight clusters then write eight output elements in parallel back to one or more output streams in the SRF. Kernels repeat this process for several loop iterations until all elements of the input stream have been read and operated on. In this manner, data-level parallelism is exploited across the eight clusters through SIMD execution and instruction-level parallelism is exploited with VLIW instructions per cluster. Locality within kernels is exploited during each loop iteration when intermediate results

are passed through the intra-cluster switch and stored in the LRFs. Producer-consumer locality between kernels is captured by storing kernel output streams in the SRF and reading input streams from the SRF during subsequent kernels without going back to external memory. While kernel execution is ongoing, the host processor can concurrently issue stream load and store instructions so that when the next kernel is ready to start, its input data is already available in the SRF.

In our example application, the stereo depth extractor, each stream in the SRF is a row from an input image, and each stream element is a pair of 16-bit pixels from that row. For each iteration of a 7x7 convolution kernel, 16 input pixels (two per cluster) are operated on, convolved with neighboring pixels, and 16 output pixels are produced. Output streams from the 7x7 convolution kernel are stored in the SRF and passed directly to the 3x3 convolution kernel. The other kernels proceed similarly. By exploiting parallelism and locality in this manner, stream processors are able to achieve high performance on a range of media applications with only modest off-chip memory bandwidth requirements.

2.3. Software System

The Imagine software system provides the compile-time and run-time support necessary for running stream programs. As shown in Figure 4, the software system includes compilers for converting StreamC and KernelC programs into host CPU assembly code and kernel microcode, respectively, and provides run-time support for issuing stream instructions to Imagine via the host CPU's external memory interface.
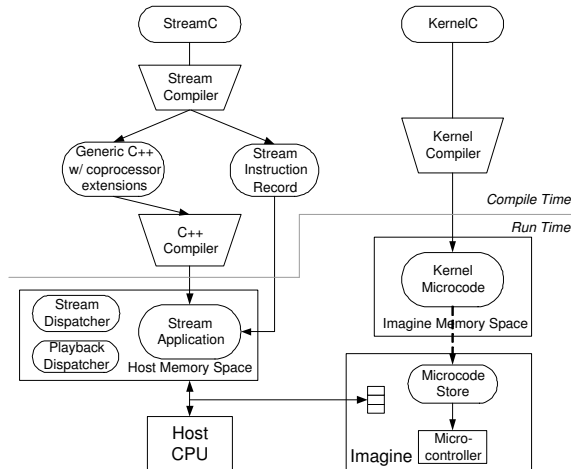
Figure 4. Imagine software system

Figure 5. Dual-Imagine development board

The KernelC compiler uses communication scheduling to produce VLIW microcode from KernelC [9]. It performs high-level optimizations such as copy-propagation, loop unrolling and automatic software pipelining, schedules arithmetic operations on functional units, specifies the data movement between ALUs and LRFs, and performs register allocation. All of the kernel microcode for an application is loaded from the host CPU into the Imagine memory space during startup. At the start of application execution, the kernel microcode is transferred from Imagine memory to the microcode store (2K VLIW instructions total). If all the kernel microcode for an application does not fit in the microcode store, the host ensures that kernels are loaded dynamically from Imagine memory to the microcode store before kernel execution occurs. If new kernels are being loaded while another kernel is being executed, a performance degradation of less than 6% occurs.

The StreamC compilation process is split into two stages. In the first stage, a stream compiler performs a number of high-level tasks such as dependency analysis between kernels and stream load/stores, software pipelining between stream loads or stores and kernel operations, optimal sizing of stripmined streams, and allocating and managing the SRF [6]. After these optimizations and analyses are completed, the stream compiler generates stream instructions (stream loads or stores, kernel invocations, and Imagine register reads and writes). These stream instructions are embedded in intermediate C++ code, which preserves the control flow of the StreamC. During the second stage, a standard C++ compiler compiles and links the intermediate code with a stream dispatcher, generating a host processor executable (assembly code).

At run time, a command line interface is used to run a StreamC application of user's choice, which inturn invokes the stream dispatcher with relevant stream instructions. The stream dispatcher manages a 32-slot scoreboard on Imagine. When a slot is free, it issues a new stream instruc-
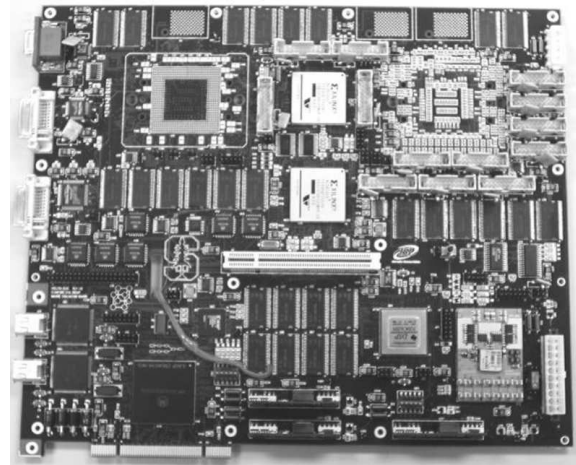
tion to be written into the scoreboard. The dependencies between the new stream instruction and other scoreboard entries are encoded with the instruction itself by the stream compiler. The stream controller on Imagine uses these dependencies to determine the next stream instruction to issue from the scoreboard when necessary resources become available. The dispatcher performs all reads/writes from/to Imagine registers, which are mapped to addresses in the host memory space. For data transfers between the host CPU and Imagine, the stream dispatcher performs memory-mapped reads/writes to a on-chip fifo.

For many media applications such as stereo depth extraction, the control-flow for the entire application is data-independent. In these cases, the StreamC compiler takes advantage of the static control-flow by using a playback mode, in which the intermediate C++ code is replaced by a record of the encoded stream instructions, in order. The playback dispatcher reads from this recorded sequence of stream instructions and dispatches them as scoreboard slots become free. Although less general than running application code on the host processor, this playback method allows for a more lightweight efficient dispatcher implementation when control-flow is data independent.

2.4. Development Board

The Imagine development board, shown in Figure 5, provides a platform for testing the Imagine stream processing hardware and software tools. The board contains two Imagine processors running at 200 MHz, fed by a single 150 MHz PowerPC 8240 host processor. Each Imagine processor has four SDRAM channels and is connected to eight 256Mbit SDRAMs running at 100 MHz for a total capacity of 256 Mbytes per Imagine. An FPGA serves as a bridge between the PowerPC and Imagine chips, providing a 66 MHz interface for issuing stream instructions and reading Imagine control registers. The board is connected to a host PC with a PCI interface, providing a command-line user interface and file storage system for the PowerPC host pro-

cessor on the development board. Additional board components are used for I/O and a multi-processor network, but are beyond the scope of this paper.

This development board provides a platform for evaluating the effectiveness of stream processing in exploiting the parallelism and locality to achieve high performance on media applications. The remainder of this paper presents this detailed evaluation using experimental measurements from this development board, except where otherwise mentioned.

3. Micro-Benchmark performance

In order to explore the range of achievable performance and power efficiency [2] on Imagine, a number of tests were written in KernelC and StreamC to exercise specific components of the Imagine stream processor. First, a set of synthetic micro-benchmarks validate the peak performance of Imagine subsystems. Second, key media processing kernels show how actual kernel characteristics affect sustained performance. Finally, another set of micro-benchmarks are used to explore how Imagine performance is affected by stream length.

3.1. Micro-benchmarks

The six micro-benchmark programs used to exercise the peak performance of Imagine components are shown in Table 1. The first four micro-benchmarks affect kernels and the last two affect application performance. Peak achieved floating-point performance while running kernels in the arithmetic clusters occurs with a mixture of floating-point adds and multiplies while peak achieved integer performance occurs with four 8-bit operations on each adder and two 16-bit operations on each multiplier. Inter-cluster communication bandwidth is stressed with a kernel that sorts 32 elements of a stream, which requires a large number of inter-cluster data exchanges, per loop iteration. Peak SRF bandwidth is achieved with a kernel that reads multiple input stream elements per loop iteration and writes the data directly back to the SRF via output streams. Memory system bandwidth and host interface bandwidth are stressed with the last two micro-benchmarks. Peak memory bandwidth is measured by running two memory stream loads that hit a small range of random memory addresses simultaneously. Host interface bandwidth is stressed by writing a series of stream instructions which continually update control registers.

Table 1 shows that all tested components except the host interface achieve over 98% of the theoretical peak. The bandwidth of the host interface is 10x lower than the theoretical peak because it is limited by the development board implementation, not the host interface on Imagine. The effect of host interface bandwidth on application performance

Component	(achieved / theoretical)	Power
Cluster (OPS)	(25.4 / 25.7) GOPS	5.79 W
Cluster (FLOPS)	(7.96 / 8.13) GFLOPS	6.88 W
Inter-cluster comm.	(7.84 / 8.00) ops/cycle	8.53 W
SRF	(12.7 / 12.8) GB/s	5.79 W
MEM	(1.58 / 1.60) GB/s	5.42 W
Host Interface	(2.03 / 20.0) MIPS	4.72 W

Table 1. Performance of Imagine components[3]

is discussed in detail in Section 5. The peak cluster performance benchmarks also demonstrate Imagine's power efficiency, sustaining 4.39 GOPS/W for integer operations and 1.16 GFLOPS/W for floating point operations. All power measurements use input data patterns that incur high toggle rates on internal nets, so during applications with highly correlated data, the average power dissipation in the stressed components will be lower.

3.2. Kernels

Although Imagine sustains close to peak performance on synthetic micro-benchmarks, when actual media processing kernels are mapped to Imagine, limited parallelism, instruction mix, and stream lengths can affect sustained performance substantially. Table 2 summarizes performance, power, register bandwidth, and instructions per cycle (IPC) for a set of kernels from the media processing and scientific computing domains. All of the kernels except for RLE and GROMACS achieve an IPC of over 35, indicating that there is ample arithmetic intensity and parallelism in these kernels to be exploited by the kernel compiler. The register bandwidth data shows that more than 95% of data accessses are to the cluster LRFs, demonstrating the large amount of locality within kernel computations. On average, most kernels require significantly less than the peak SRF bandwidth of 12.71 GB/s, meaning that some headroom remains for memory streams and for portions of kernels requiring bursts of higher SRF bandwidth.

Further analysis provides insight into the differences between peak and sustained performance on kernels. Runtime during kernel execution can be classified into the four categories shown in Figure 6: minimum inner-loop run-time required to execute arithmetic operations, extra run-time within kernel inner loops incurred from ILP limitations and load imbalance, non-main loop execution time, and cluster stalls.

The first two categories combined from Figure 6 is the percentage of execution time spent in kernel main loops. 'Operations' are what execution time would be if peak performance were achieved within the main loop (operations) and the remaining execution time is due to limited ILP within cluster main loops and load imbalance between the types of arithmetic units in a cluster. For example, the inner loop of the *update2* kernel executes inner products requiring one multiplication and one addition per element. Since the Imagine clusters have 3 adders and 2 multipliers, per-

---

Kernel	ALU	LRF BW (GB/s)	SRF BW (GB/s)	IPC	Power (Watt)	
2D DCT	6.92 GOPS	95.13	2.09	35.9	6.99	two-dimensional direct cosine transform of 16-bit 8-by-8 pixel macroblocks
blocksearch	9.62 GOPS	98.18	0.07	35.9	7.07	search similar macroblocks for motion estimation
RLE	1.21 GOPS	54.00	0.39	19.8	6.15	apply run length encoding to macroblocks (16 bit)
conv7x7	9.76 GOPS	99.44	2.75	36.7	7.76	convolve images with a 7x7 filter (16 bit)
blockSAD	4.05 GOPS	108.7	8.87	35.2	7.79	compute SAD of two images (16 bit)
house	3.67 GFLOPS	101.4	2.26	35.3	7.05	compute the Householder matrix (float)
update2	5.82 GFLOPS	119.6	2.20	47.5	7.42	matrix-matrix multiplication (float)
GROMACS	2.24 GFLOPS	44.18	0.51	15.3	6.65	force computation between water molecules (float)

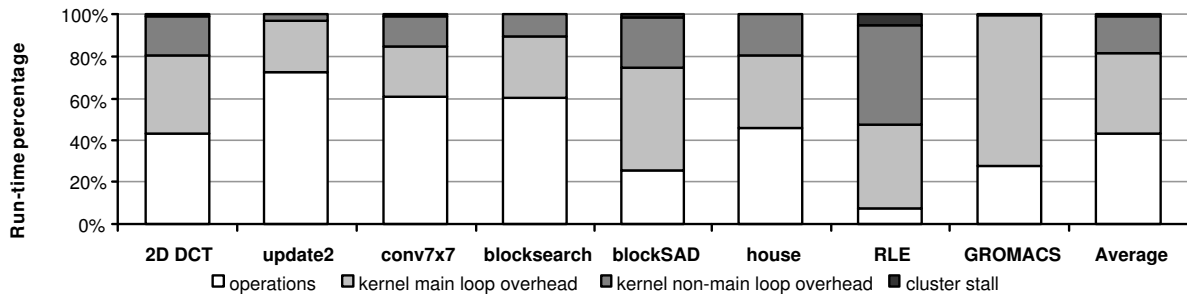Table 2. Performance of representative kernels of media applications on Imagine

Figure 6. Breakdown of kernel performance

formance in this case is limited by the multiplication units. GROMACS and RLE on the other hand are two kernels which have even worse main-loop performance relatively because they are limited by divide/square-root and scratch-pad bandwidth, respectively. Dependencies between arithmetic operations limit ILP in other kernels, another cause of degradation in main-loop performance.

The third and fourth categories in the kernel run-time percentage are due to non-main loop cycles and cluster stalls. Non-main loop cycles are cycles during kernel execution spent in loop prologues, epilogues, outer loop blocks, and in additional main loop iterations used to prime software pipelined loops. Note that when kernels are invoked to operate on longer streams, more main loop iterations are executed and the percentage of run time in non-main loop cycles goes down. In Figure 6, average stream lengths measured during application execution were used to measure the run-time percentages. The effect stream length has on kernel performance is analyzed in detail in the next section. The last category of run-time percentage accounts for cluster stalls during kernel execution when the SRF is not ready to accept new stream read or write requests from the clusters. Since the average SRF bandwidth used by these kernels is significantly lower than the peak SRF bandwidth, these cluster stalls occur during kernel startup periods when SRF streams have not been initialized and during kernels which have bursty SRF bandwidth requirements. However, even in these kernels, less than 5% of kernel run-time cycles are spent in stalls waiting for the SRF.

3.3. Stream Length Effects

As shown in the previous section by the percentage of kernel runtime spent in non-main loop cycles, stream lengths affect the average performance of kernels. Stream lengths also affect average memory bandwidth since loads and stores of long streams can more easily hide the latency of accessing individual stream elements from external memory. In this section, we present a set of synthetic micro-benchmarks to analyze the effect of stream length on both kernel performance and memory performance.

The first two sets of micro-benchmarks demonstrate the effect of stream length on kernel performance. Both micro-benchmarks make use of a kernel where the main loop sustains 4.8 GOPS and the non-main loop portion sustains 1.6 GOPS. Furthermore, the main loop is software pipelined in order to achieve higher arithmetic intensity and to simulate typical kernel conditions. Average performance is measured over a time period when this kernel is repeatedly issued from the host in order to make sure host interface bandwidth effects are taken into account.

The first micro-benchmark, shown in Figure 7, keeps the prologue length fixed at 64 cycles and varies the stream length for different sized main loops. Since this kernel reads one stream element per cluster per loop iteration, in this micro-benchmark the number of main loop iterations executed per kernel invocation is 1/8 the stream length. The ideal BW number corresponds to an infinite stream length, when all execution time would be spent in the main loop. As shown in the figure, shorter streams degrade the perfor-
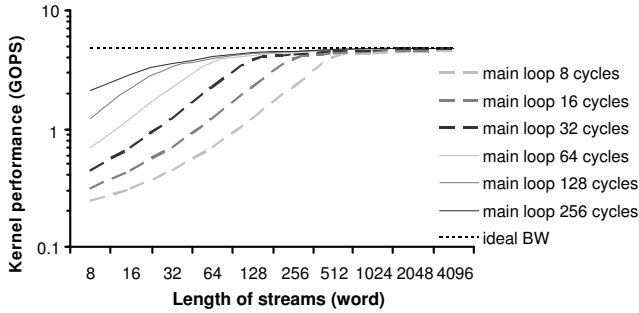
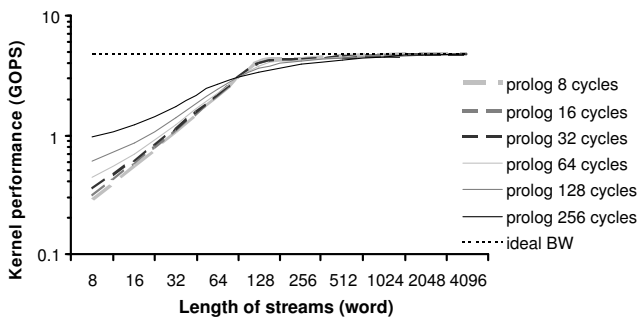Figure 7. Kernel performance with varied stream length - prologue fixed at 64 cycles

Figure 9. Memory system performance from a single AG with varied stream lengths

Figure 8. Kernel performance with varied stream length - main loop fixed at 32 cycles

Figure 10. Memory system performance from two AGs with varied stream lengths

mance of the shorter main loop kernels more severely because a larger percentage of kernel run-time is spent in non-main-loop cycles. Also, note that there are cases when the arithmetic performance of the kernel is less than 1.6 GOPS, the performance of the non-main-loop portion of the kernel. This is because it takes more time to send stream instructions from the host to run a kernel than to run a kernel in some cases, meaning the clusters are idle waiting for the next kernel to be issued. On the development board, it takes about 500 ns to send a stream instruction over the host interface and this kernel requires 5 stream instructions before it can start running, so that if the kernel execution time is less than $2.5\mu s$, kernel performance is limited by the host interface bandwidth.

In the second micro-benchmark, shown in Figure 8, the main loop cycles are fixed and the effect of stream length is measured for different prologue lengths. In this case, for stream lengths of 64 elements or less, kernel performance is dominated by limited host bandwidth. As a result, kernels with shorter prologues have worse performance because the clusters spend a larger percentage of time idle waiting for the next kernel to be issued. For stream lengths greater than 64, performance is dominated by the percentage of run time in main-loop vs. non-main-loop cycles, so shorter prologues have higher performance.
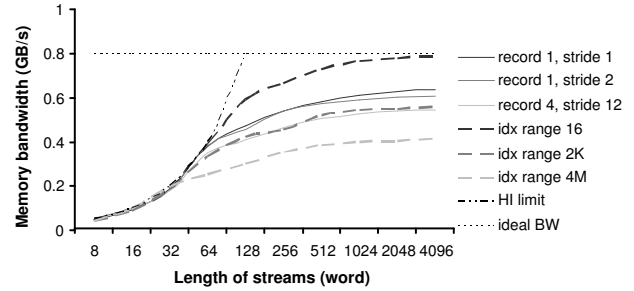
The next two micro-benchmarks demonstrate the effect of stream length on the memory system performance. On Imagine, there are two address generators (AGs) in the memory system, allowing two concurrent stream load or store instructions to be executed. The first micro-benchmark, shown in Figure 9, plots the memory system performance as stream length is varied when only one of these AGs is used at a time. In this experiment, six different memory access patterns - unit stride with record size one, stride 2 with record size one, stride 12 with record size 4, indexing random addresses over the range of 16 words to 2K words and 4M words - are used since external DRAM bandwidth is heavily influenced by the sequences of addresses received.

For stream lengths of less than 64, on most access patterns the memory system bandwidth is still limited by the host interface bandwidth, even though each memory access has 30 to 40 cycles of latency. As streams get longer, the achieved memory bandwidth approaches the limits of the external DRAM. The indexed random with range of 16 words pattern is a special case where all memory accesses are captured in a small on-chip cache in the Imagine memory controller, thus stressing the on-chip maximum AG bandwidth rather than the maximum external DRAM bandwidth. The second memory system micro-benchmark

Appli-cation	ALU	IPC	Performance Summary	Power (Watts)
DEPTH	4.91 GOPS	33.3	90 frames/s	7.49
MPEG	7.36 GOPS	31.7	138 frames/s	6.80
QRD	4.81 GFLOPS	40.1	326 QRD/s	7.42
RTSL	1.30 GOPS	17.7	44 frames/s	5.91

Table 3. Application Performance

is shown in Figure 10. It is similar to the above micro-benchmark, but with two simultaneously active AGs. As a result, higher bandwidth is achieved in some patterns when there are no DRAM bank conflicts between the two memory streams.

Note that although indexed memory addresses over a small range approach the theoretical peak of 1.6 GB/s asymptotically, the unit stride case is approximately 20% lower than was expected from simulation. This discrepancy is due to a performance bug in the on-chip memory controller which causes unnecessary DRAM precharges between some accesses to the same DRAM row.

In summary, the micro-benchmarks presented in this section show how Imagine is able to achieve near peak performance with synthetic micro-benchmarks. However, both the characteristics of typical media processing kernels and stream length effects lead to a difference between sustained performance and peak performance during the execution of real applications.

4. Application Performance

The previous section studied the performance of specific components of Imagine in isolation. In this section we will study the performance of entire applications, which depends on all of these components working together.

4.1. Overview

Table 3 lists the overall performance for four applications: DEPTH is the stereo depth extractor first presented in Section 2; MPEG encodes three frames of 360x288 24-bit video images according to the MPEG-2 standard; QRD converts a 192x96 complex matrix into an upper triangular and an orthogonal matrix, and is a core component of space-time adaptive processing [1]; and RTSL renders the first frame of the SPECviewperf 6.1.1 advanced visualizer benchmark using the Stanford Real-Time Shading Language [10].

The first column in the table, which lists the number of arithmetic operations executed per second, shows that Imagine sustains up 7.36 GOPS and 4.81 GFLOPS. If we consider all operations, not just arithmetic ones, then Imagine is able to sustain over 40 instructions per cycle on QRD. In fact, for all three video applications, Imagine can easily meet real-time processing demands of 24 or 30 fps. These high absolute performance numbers are a result of carefully managing bandwidth, both in the programming model and
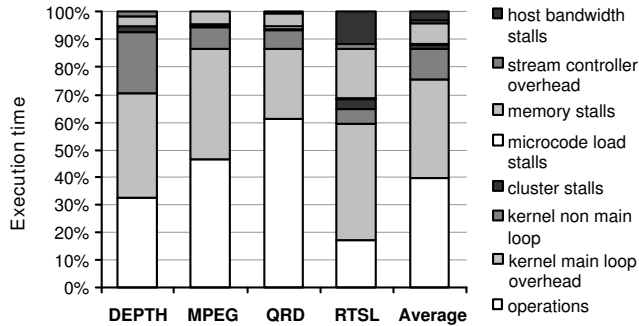

Figure 11. Execution time breakdown of applications obtained from cycle-accurate simulation.

in the architecture, as discussed in Section 2. Imagine dissipates between 5.9W and 7.5W while executing these applications at their highest performance levels. Furthermore, as shown in [7], voltage and frequency scaling allow the same Imagine chip to execute the MPEG and QRD applications at about half the performance but only one-fourth the power ($< 2W$).

4.2. Analysis of Overheads

When compared to the peak capabilities of Imagine, the applications achieve between 16% to 60% of the maximum arithmetic performance, and between 21% to 50% of the maximum IPC. The difference between the peak performance of Imagine and the achieved performance is due to several factors, shown graphically in Figure 11. The first four categories account for kernel run-time in the clusters, comprising 90% of the execution time for all applications except RTSL. The kernel run-time, as discussed in Section 3.2, is affected by operation mix, limited ILP, average stream lengths, and required SRF bandwidth in the kernels. During the remaining execution cycles, the clusters idle waiting for a new kernel invocation due to one of four overheads: stalls waiting for a microcode load operation to complete, memory stalls (waiting for a stream load or store to complete), stream controller issue overhead incurred once per stream instruction, and stalls due to inadequate host interface bandwidth. If more than one of these is responsible for delaying a kernel at any given time, the extra cycles are attributed to the overhead earliest in the list.

While these four non-kernel overheads occupy less than 10% of the total execution time for the first three applications, they occupy over 30% of the time for RTSL. The two biggest culprits are memory stalls and host interface stalls. The large overhead caused by memory stalls may come as a surprise since RTSL actually requires the least DRAM bandwidth of any of the presented applications (see Section 5.1). It turns out that the problem is not pure bandwidth. Instead, there are periods in the application where kernels are required to wait until a memory load or store finishes, either because the kernel needs to use the data still be-
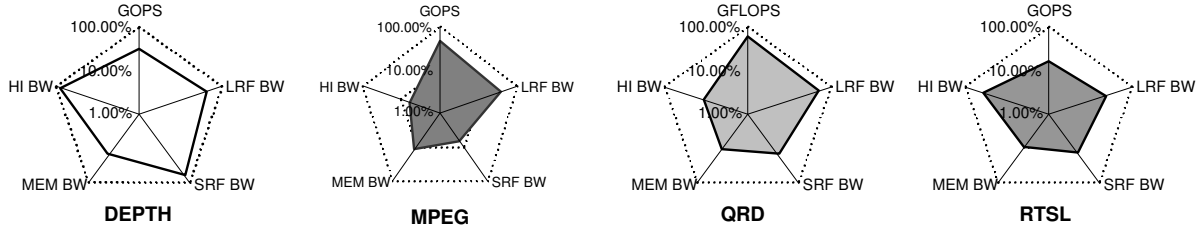
Figure 12. Average sustained performance of Imagine components on applications

ing loaded or because it needs to use the SRF space that is occupied by a stream being stored to memory. The stream compiler does not do as good a job on scheduling memory operations and kernel operations concurrently for RTSL as it does with the other applications, mainly because stream lengths in RTSL are less predictable from batch to batch. The other major non-kernel overhead for RTSL, host interface stalls, arises when control-flow decisions on the host processor are serialized on kernel results, causing Imagine to idle during a Imagine-host processor round-trip delay. While certain portions of these non-kernel overheads are probably inherent to the RTSL application, future research can produce superior compilation strategies to overlap Imagine operations and improve kernel occupancy to higher than 70%.

Further insight into application performance is provided by measuring the average performance of Imagine components during application execution. Figure 12 shows the percentage of peak GOPS, host interface (HI) bandwidth, memory bandwidth, SRF bandwidth, and LRF bandwidth sustained on the four applications plotted on a logarithmic scale. This data demonstrates a range of application characteristics typical to media applications and shows that different applications stress different components of Imagine. The effect that these four components - the host interface, the memory system, SRF, and LRFs - have on arithmetic performance are explored in more detail in the next section.

5. Discussion

5.1. Arithmetic to Memory Bandwidth Ratio

Imagine's arithmetic to memory bandwidth ratio for floating point computation is over 20:1. It can perform over 20 floating point operations for each 32-bit word transferred over the memory interface. This is $5\times$ higher than the 4:1 ratio typical of conventional microprocessors and DSPs [14, 16]. Yet Imagine is still able to sustain half of peak performance on a variety of applications. Many conventional processors do not do as well even with their greater memory bandwidth.

Imagine is able to achieve good performance with relatively low memory bandwidth for two reasons. First, exposing a large register set with two levels of hierarchy to the compiler enables considerable locality (kernel locality and producer-consumer locality) to be captured that is not cap-
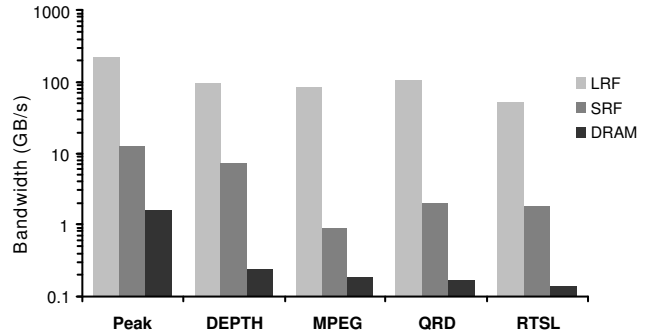

Figure 13. Bandwidth hierarchy of applications

tured by a conventional cache. This locality is evidenced by the LRF to memory bandwidth ratio of over 350:1 across four applications shown in Figure 13.

Second, by scheduling stream loads and stores to hide latency, the Imagine memory system can be designed to provide the *average* bandwidth required by applications without loss of performance. In contrast, conventional processors are highly sensitive to memory latency and hence providing memory bandwidth well in excess of the average is required to avoid *serialization latency* on each memory access. Although the memory system of a conventional processor is idle much of the time, reducing its bandwidth would increase memory latency (by the additional cycles required to transfer a cache line across the lower bandwidth interface) and hence increase execution time.

5.2. Kernel versus Application Evaluation

Some previous studies of stream processors [15, 2] showed relatively low performance. These results were due to evaluating the performance of a single kernel accessing its input and output streams from memory. The bandwidth demands of many kernels exceed the sustainable memory bandwidth of Imagine (see the SRF BW column of Table 2). Thus, if run from memory, these kernels would be memory bound. However, in actual applications, most kernels run from the SRF, not from memory, and the scheduler uses software pipelining to hide the latency of the few required memory operations under the execution time of many kernels. The results in Figure 13 show conclusively that for real applications, a stream processor is not memory bound.

| Application | Stream Ops | | Register Ops | | | | | SDR References | | BW |
	Kernel + Restart	Memory	SDR Write	MAR Write	UCR Write	Move	Misc.	(Reuse)	Total	(MIPS)
DEPTH	5344	3629	45	3608	4635	0	11	32263 (717×)	17272	1.6
MPEG	489	407	182	412	528	108	428	2744 (15.1×)	2554	0.12
QRD	252	103	215	93	36	0	6	664 (3.09×)	705	0.23
RTSL	2835	1535	6477	1536	1733	927	1643	15166 (2.34×)	16685	0.75

Table 4. Histogram of stream operation lists of media applications

	Avg kernel duration	Avg kernel stream length	Avg memory stream length
DEPTH	374.5 cycles	606.3 words	188.6 words
MPEG	8244 cycles	1191 words	2543 words
QRD	2234 cycles	2087 words	1261 words
RTSL	1022 cycles	585.8 words	520.0 words

Table 5. Cluster characteristics of applications

There is a serious pitfall involved in evaluating processors, and in particular stream processors, using just kernels. As described in Section 5.1, much of the efficiency of a stream processor derives from its ability to (a) forward results from kernel to kernel via the SRF, and (b) to hide memory latency by overlapping memory references with the execution of *several* kernels, not just one. Both of these advantages are lost in a kernel-only evaluation.

5.3. Stream Instruction Analysis

The different stream instruction types supported by Imagine are shown in Table 4 in the form of an histogram per application. "Stream Ops" either transfer or process entire data streams. This includes kernel operations, memory loads and stores, as well as cluster "Restart" operations that are used to sequence multiple finite streams between the SRF and clusters while presenting the abstraction of a single longer stream to the kernel. "Register Ops" read or write control registers. These can be used to pass parameters to kernels ("UCR Write") and access descriptor registers that hold length and location information for streams in the SRF ("SDR Write") and in DRAM ("MAR Write"). Since the data held in the descriptors consists of a large number of bits, these registers help to reduce the required amount of host instruction bandwidth. Instead of encoding the length and location information for streams into all stream instructions, we can instead send the information once via a descriptor write and then use it multiple times by having other operations simply refer to the descriptor index. Imagine contains 32 SDRs and 8 MARs. Data can also be transferred between register files ("Move"). Finally, there are other stream instruction types for tasks such as loading microcode and synchronization between different Imagine components. These have been lumped into a single category on the chart ("Misc.").

The stream instruction bandwidth used by each application is shown in the last column in Table 4. DEPTH required the most bandwidth, 1.6 MIPS, because it operates on relatively short streams (see Table 5). Other applications require less than half the 2 MIPS maximum bandwidth possible in our current setup. It is interesting to note the number of times each SDR is reused. If it were not for the 717× reuse of SDRs, it is likely that DEPTH would exceed the maximum possible host bandwidth. For example, if only the minimum amount of SDR reuse was achieved, i.e., 2×, then the total number of stream instructions would increase by 1.9x. DEPTH does so well at reusing SDRs because all possible SDR values fit within the 32 entries in the SDR file, reducing the necessary host interface bandwidth. However, a change in the implementation of DEPTH might require more SDR values than can fit in the SDR file, resulting in performance losses due to insufficient host bandwidth. To alleviate this host bandwidth bottleneck, a new evaluation board is being built that is expected to achieve closer to the peak theoretical host bandwidth of 20 MIPS. This impact of host bandwidth is quantified in the following subsection.

5.4. Host Processor Bandwidth

Each kernel execution, memory operation, or scalar register update on Imagine requires that the host processor transfer a stream instruction to the stream controller. The 32-entry scoreboard in the stream controller acts as a buffer, allowing the host processor to run ahead of Imagine, buffering up instructions for future use. Limited host interface bandwidth can affect Imagine performance in two ways. First, if the average demand for stream instructions is greater than the host bandwidth, the scoreboard will ultimately empty, causing Imagine to idle waiting for new instructions. Second, some applications require that the host read the results of a previous stream instruction before issuing the next instruction. These host dependencies can stall execution until a host read-compute-write cycle is completed.

We use the most demanding application, DEPTH, to evaluate the effect of varying host interface bandwidth. Figure 14 shows how execution time is affected as host interface bandwidth is varied from 0.5 MIPS to 50 MIPS. The figure shows that with a host bandwidth of 2 MIPS or greater, Imagine never idles waiting on the host. With bandwidth less than 2 MIPS, execution time increases with the inverse of bandwidth. Most of the increased execution
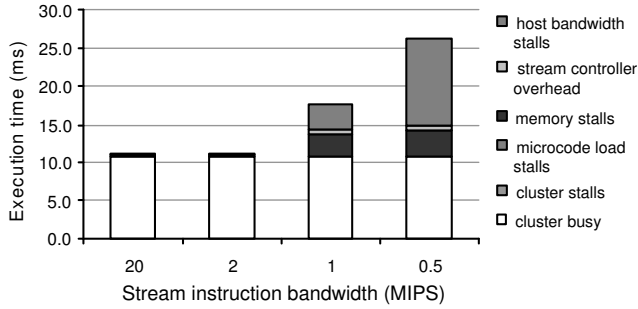
Figure 14. Execution time breakdown on DEPTH over different host interface bandwidth

App	Lab running cycles	ISIM running cycles
DEPTH	2.22 M	2.11 M
MPEG	4.33 M	4.24 M
QRD	0.615 M	0.603 M
RTSL	4.47 M	4.24 M

Table 6. Lab vs. ISIM running cycles

time is spent waiting on the host interface ("host bandwidth stalls"). Also, with reduced host bandwidth, it is not possible to fully overlap memory operations with kernel execution, which results in the clusters idling waiting on a memory operation to complete ("memory stalls").

5.5. Simulation versus Experiment

As shown in Table 6, execution times measured in the laboratory are within 6% of the results measured from the Imagine cycle accurate simulator, ISIM. In all cases, the actual hardware is slower than simulation. These differences are due to small inaccuracies in the simulation model. In the hardware, pipeline latencies not modeled in ISIM are incurred when kernels and stream loads or stores are issued from the stream controller. Furthermore, a performance bug in the hardware implementation of the memory controller results in lower sustained memory bandwidth under certain memory access patterns. Finally, ISIM's model of host processor execution is optimistic, resulting in lower simulated execution times for host processor dependencies. The effect of these seemingly innocuous differences on overall execution time highlights the need to validate all simulation results against actual hardware.

5.6. Power Efficiency Comparison

Imagine compares very favorably on power efficiency to other programmable floating-point processors when normalized to the same technology generation. As explained in Section 3.1, Imagine achieves a performance efficiency of 1.16 GFLOPS/W (862 pJ per floating-point operation) at 1.8 Volts on the peak cluster performance benchmark. When normalized to a modern 0.13 μm 1.2 Volt process technology, Imagine would achieve 277 pJ/FLOP on this

benchmark, between 3x and 13x better than modern commercial programmable DSPs and microprocessors targeted for power efficiency in similar process technologies. For example, based on its published peak performance and power dissipation in a 0.13 μm 1.2 Volt technology, the 225 MHz TI C67x DSP dissipates 889 pJ/FLOP [16] while the 1.2 GHz Pentium M dissipates 3.6 nJ/FLOP [4]. Furthermore, the improved design methodologies and circuit designs typically used in these commercial processors would provide additional improvement in the achieved power efficiency and performance of Imagine, demonstrating the potential of stream processors to provide over an order of magnitude improved power efficiency when compared to commercial programmable processors.

6. Conclusion and Future Direction

This paper has presented an experimental evaluation of the Imagine stream processor. A set of microbenchmarks was used to characterize the performance of each component of Imagine. These tests show that except for host interface bandwidth, the Imagine components individually are able to achieve 98% of their theoretical peak.

A set of *kernels* was run to determine how these components work together on small blocks of code. These tests show that Imagine achieves an average of 43% of peak performance across a number of kernels. The difference between peak and achieved performance at the kernel level is due to time spent in loop prologues/epilogues and unused instruction slots due to limited ILP or load imbalance across types of arithmetic units.

Four full Imagine applications were evaluated to measure the effects of memory operations, host bandwidth, and inter-kernel communication on performance. Imagine sustains between 16% and 60% of the peak arithmetic performance and between 21% and 50% of the peak IPC on these applications. Much of the difference between peak and achieved application level performance is due to the kernel issues discussed above. At the application-level a small amount of time is lost due to kernels waiting on the memory system or the host interface. For all of our applications except RTSL, this application-level overhead was less than 10% of execution time. RTSL has an application-level overhead of 30% due to dependences with the host processor that serialize execution.

Overall, our evaluation has shown that the Imagine stream processor can efficiently support a wide variety of applications (ranging from molecular dynamics to video compression). Programming entirely in a high-level language, with no assembly performance tuning, stream and kernel compilers are able to keep an array of 48 floating-point-units busy using a combination of data-level parallelism (8-way SIMD) and instruction-level parallellism (6-way VLIW) - achieving over 50% utilization at the kernel level (and over 80% on many inner loops). The same compilation tools are able to efficiently ex-

ploit the two-level register hierarchy of Imagine keeping the register to memory ratio over 350:1 across all four applications.

ASICs are becoming more difficult and costly to design. At the same time, many media applications demand the flexibility of a programmable processor — for example to support multiple video codecs or multiple wireless air interfaces. Stream processors address these issues by providing performance competitive with an ASIC while retaining the flexibility of a programmable processor.

Stream processors pose many opportunities and challenges for compilers. Our current software system addresses some of these — using communication scheduling [9] to schedule kernels on clusters with partitioned LRFs and using stream scheduling [6] to schedule stream operations and allocate the SRF. Much more can be done, however, to optimize the mapping of programs onto a stream architecture with an exposed register hierarchy. We expect future stream compilers to restructure applications by splitting and joining kernels to better use the available LRF and SRF bandwidth. Better stream scheduling algorithms will optimize re-use in the SRF and minimize kernel stalls on memory operations. Also, we expect techniques to be developed that automate the extraction of kernels and streams from conventional "C" code, effectively converting "C" into StreamC and KernelC.

Recent work has shown that stream processing is applicable to scientific computing [3]. Good performance has been demonstrated on codes with both irregular and regular grids. Streams have also been applied to network processing [11] and software defined radios [12]. Efficient streaming demands only large amounts of data paralellism and a high arithmetic to memory ratio, not regularity. We expect many other applications of stream processing to emerge.

7. Acknowledgements

References

[1] K. C. Cain, J. A. Torres, and R. T. Williams. RT_STAP: Real-time space-time adaptive processing benchmark. Technical Report MTR 96B0000021, MITRE, February 1997.

[2] S. Chatterji, M. Narayanan, J. Duell, and L. Oliker. Performance evaluation of two emerging media processors: Viram and imagine. In *International Parallel and Distributed Processing Symposium*, pages 229–235, April 2003.

[3] W. J. Dally, P. Hanrahan, M. Erez, T. J. Knight, F. Labonté, J. H. Ahn, N. Jayasena, U. J. Kapasi, A. Das, J. Gummaraju, and I. Buck. Merrimac: Supercomputing with streams. In *SC2003*, November 2003.

[4] Intel®. Intel® pentium® m processor. http://www.intel.com/design/mobile/datashts/25261202.pdf.

[5] T. Kanade, A. Yoshida, K. Oda, H. Kano, and M. Tanaka. A stereo machine for video-rate dense depth mapping and its new applications. In *Proceedings of the 15th Computer Vision and Pattern Recognition Conference*, pages 196–202, San Francisco, CA, June 18–20, 1996.

[6] 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.

[7] U. J. Kapasi, S. Rixner, W. J. Dally, B. Khailany, J. H. Ahn, P. Mattson, and J. D. Owens. Programmable stream processors. *IEEE Computer*, pages 54–62, August 2003.

[8] 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.

[9] 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, November 2000.

[10] K. Proudfoot, W. R. Mark, S. Tzvetkov, and P. Hanrahan. A real-time procedural shading system for programmable graphics hardware. In *Proceedings of ACM SIGGRAPH*, pages 159–170, August 2001.

[11] J. S. Rai. A feasibility study on the application of stream architectures for packet processing applications. Master's thesis, North Carolina State University, Raleigh, NC, 2003.

[12] S. Rajagopal, S. Rixner, and J. R. Cavallaro. A programmable baseband processor design for software defined radios. In *45th IEEE International Midwest Symposium on Circuits and Systems*, volume 3, pages 413–416, August 2002.

[13] S. Rixner. *Stream Processor Architecture*. Kluwer Academic Publishers, Boston, MA, 2001.

[14] D. Sager, G. Hinton, M. Upton, T. Chappell, T. D. Fletcher, S. Samaan, and R. Murray. A 0.18μm CMOS IA32 microprocessor with a 4GHz integer execution unit. In *2001 IEEE International Solid-State Circuits Conference Digest of Technical Papers*, pages 324–325, February 2001.

[15] J. Suh, E.-G. Kim, S. P. Crago, L. Srinivasan, and M. C. French. A performance analysis of pim, stream processing, and tiled processing on memory-intensive signal processing kernels. In *30th Annual International Symposium on Computer Architecture*, pages 410–421, June 2003.

[16] Texas Instruments. *TMS320C6713 Floating-Point Digital Signal Processors*, 2003.03 edition.