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

The Imagine Stream Processor

Ujval J. Kapasi, William J. Dally, Scott Rixner†, John D. Owens, and Brucek Khailany *

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

†Computer Systems Laboratory
Rice University, Houston, TX 77005, USA
rixner@rice.edu

Abstract

The Imagine Stream Processor is a single-chip programmable media processor with 48 parallel ALUs. At 400 MHz, this translates to a peak arithmetic rate of 16 GFLOPS on single-precision data and 32 GOPS on 16-bit fixed-point data. The scalability of Imagine's programming model and architecture enable it to achieve such high arithmetic rates. Imagine executes applications that have been mapped to the stream programming model. *The stream model decomposes applications into a set of computation kernels that operate on data streams. This mapping exposes the inherent locality and parallelism in the application, and Imagine exploits the locality and parallelism to provide a scalable architecture that supports 48 ALUs on a single chip. This paper presents the Imagine architecture and programming model in the first half, and explores the scalability of the Imagine architecture in the second half.*

1. Introduction

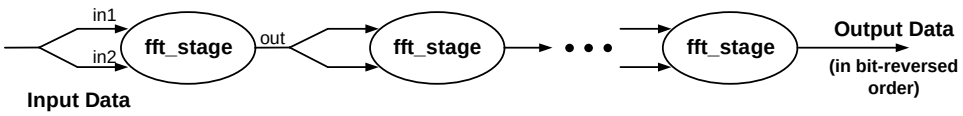
Media-processing applications, such as 3-D polygon rendering, MPEG-2 encoding, and stereo depth extraction, are becoming an increasingly dominant portion of computing workloads today. The real-time performance constraints of these applications coupled with high arithmetic intensity require parallel solutions that can scale to meet these demands. Fortunately, media-processing applications inherently contain a large amount of data-parallelism. Furthermore, providing large numbers of ALUs to operate on data in parallel is relatively inexpensive. In today's VLSI technology, hundreds of 32-bit adders can fit on a single 1 cm^2 chip. Yet current programmable solutions cannot scale to support this many ALUs, even though by themselves the ALUs easily fit on a chip. This is because both providing instructions and transferring data at the necessary rates are problematic. For example, a 48 ALU single-chip processor must issue up to 48 instructions/cycle and provide up to 144 words/cycle of data bandwidth to operate at peak rate.

The *Imagine Stream Processor* addresses these issues by using the stream programming model to expose parallelism as well as producer-consumer locality, the true data locality in media processing applications. This locality can be exploited by routing most of the required bandwidth on local wires, which are more efficient and plentiful than global communication paths. Imagine exploits this locality with a bandwidth hierarchy that sustains the data bandwidth necessary to support 48 ALUs on a single die. This translates to a peak of 16 GFLOPS on single-precision applications and 32 GFLOPS on 16-bit applications. On a variety of realistic applications, as shown in Table 1, Imagine can sustain up to 50 instructions per cycle, and up to 15 GOPS of arithmetic bandwidth. In order to evaluate the scalability of the architecture, a prototype was built that contains the maximum number of ALUs that could be supported on a single chip. Early studies of what the targeted technology could support led to the choice of 48 ALUs for the Imagine prototype. The 2.6 cm^2 prototype was developed in collaboration with Texas Instruments, in their 1.5V 0.15μm CMOS process. The Imagine prototypes are being tested and debugged

Table 1: Performance of 16-bit and floating point applications on Imagine

Applications (16-bit in italics)	Arithmetic		All Operations	
	IPC	GOPS	IPC	% of peak
Depth Extraction [3]	*30*	*11.9*	*50*	*45%*
MPEG-2 Encode	*38*	*15.4*	*53*	*47%*
QR-decomposition	26	10.5	44	61%
Render (sphere) [8]	15	5.9	28	40%

*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.



(a) Stream and kernel representation

```
void fft(stream<complex>     in,
         stream<complex>[10] twiddle,
         stream<complex>     out)
{
  tmp1 = in;  // does not imply copy
  for (int i=0; i<10; i++) {
    fft_stage(tmp1[0..511], tmp1[512..1023],
              twiddle[i], tmp2);
    tmp1 = tmp2;
  }
  out = tmp1.bitrev();
}
```

(b) Stream-level program pseudocode

```
kernel fft_stage(istream<complex> in1,
                 istream<complex> in2,
                 istream<complex> twiddle,
                 ostream<complex> out)
{
  for (int i=0; i<512; i++) {
    in1 >> a;
    in2 >> b;
    twiddle >> W;
    out << (a+b) << (W*(a-b));
  }
}
```

(c) Kernel-level program pseudocode

Figure 1: 1024-point complex radix-2 FFT.

in the lab (July 2002). So far, simple kernels and memory transfers have been tested without flaws. A fully-featured dual-Imagine development board is concurrently being debugged in order to facilitate the testing of more complex applications. Final static timing analyses reported 296 MHz for the prototype in the typical process corner. However, since the parts are expected to function up to 400 MHz in the lab, this clock rate is assumed for all results in this paper.

The remainder of this paper is organized into two major parts. The first part presents an overview of the stream programming model using FFT as an example (Section 2) and then presents the Imagine architecture (Section 3). The next part of the paper discusses the scalability of the Imagine architecture to a large number of ALUs. This includes a discussion, in Section 4, of how to organize the various modules on Imagine to provide enough instruction and data bandwidth for the increased number of ALUs. Section 5 then illustrates some of these concepts with examples of real kernels and applications. A brief section on different but related approaches follows (Section 6). The last section summarizes the conclusions drawn in the paper (Section 7). Finally, further details on the Imagine architecture and its programming model and languages can be found in [6].

2. An Example Stream Application: FFT

The Imagine Stream Processor executes applications that have been mapped to the stream programming model. This programming model organizes the computation in an application into a sequence of arithmetic *kernels*, and organizes the data-flow into a series of data *streams*. The data streams are ordered, finite-length sequences of data records of an arbitrary type (although all the records in one stream are of the same type). The inputs and outputs to kernels are data streams. The only non-local data a kernel can reference at any time are the current head elements of its input streams and the current tail elements of its output streams.

A simple example, in Figure 1a, shows the mapping of a 1024-point radix-2 FFT to the stream model. Each oval in the figure corresponds to the execution of a kernel, while each arrow represents a data stream transfer. In this case, each stream is composed of complex floating point elements. The overall FFT requires 10 butterfly stages, and thus 10 calls to the *fft_stage* kernel. In the stream implementation, the data are read and written according to a perfect shuffle pattern [9], so the kernel requires two input streams and one output stream. The output of the last kernel is in bit-reversed order, so it must be reordered through memory. Finally, a third input stream, not shown in Figure 1a, provides the necessary twiddle factors required for each stage.

In order to execute the stream version of FFT, two types of programs, which operate at different levels, are necessary. One program, shown in Figure 1b, will operate at the *stream level* and will control both the order of kernel execution and the transfer of data streams for each kernel. The other type of program, shown in Figure 1c, will operate at the *kernel level* and will dictate the actual computation that must occur on each stream element. Applications that are more involved than the FFT example map to the stream model in a similar fashion. Examples can be found in other references: Khailany et al. discuss the mapping of a stereo depth extractor [6]; Rixner discusses the mapping of an MPEG-2 encoder [10]; and Owens et al. discuss the mapping of a polygon rendering pipeline [8].

The stream model is important because it organizes an application to expose the locality and parallelism information

that is inherent in the application. Locality is exposed both within a kernel — temporaries such as the result of `(a-b)` are guaranteed to stay local to the kernel — and producer-consumer locality between kernels is explicit from the stream graph. This ensures that the only data transferred between the clusters and SRF are records that need to be passed from kernel to kernel, and that the only data transferred between the SRF and DRAM are records that are part of truly global data structures. In FFT, for example, only data elements passed between stages need to access the SRF, and only the initial input data and final output data need to access the global memory space in DRAM. Data-level parallelism is exposed between stream elements since the butterfly calculation in *fft_stage* could be applied to all 512 complex pairs in parallel. Also, instruction-level parallelism is available between operations in a kernel — `(a+b)` and `(W*(a-b))` can be calculated in parallel — and task-level parallelism exists between kernels in an application.

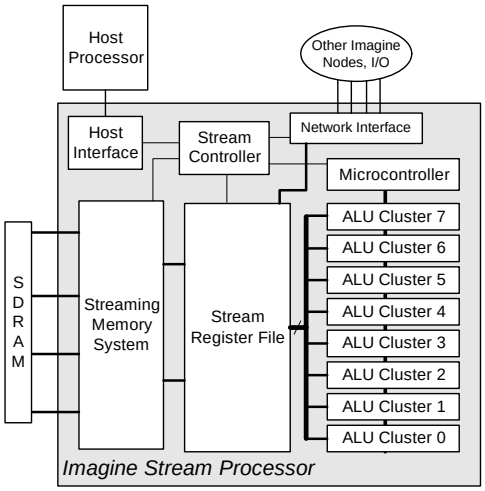


Figure 2: Imagine architecture block diagram

3. Imagine

Imagine is a programmable stream processor and is a hardware implementation of the stream model. A block diagram of the architecture is shown in Figure 2. Imagine is designed to be a stream coprocessor for a general purpose processor that acts as the host, as seen in the figure. The stream controller sequences stream commands from the host processor and issues them to the various modules on the chip. All data stream transfers are routed through a 32 KW stream register file (SRF). The streaming memory system transfers entire streams between the SRF and off-chip SDRAM. Kernel programs consist of a sequence of VLIW instructions and are stored in a 2K $\times$ 576-bit RAM in the microcontroller. The microcontroller issues kernel instructions to eight arithmetic clusters in a SIMD manner. Each cluster consists of six ALUs (plus a few utility functional units) and 304 registers in several local register files (LRFs). The network interface module routes streams between the SRF of its node and the external network.

3.1. Stream-level ISA

Table 2 presents an abridged list of the stream-level instructions supported by Imagine; operations not listed in the table include scalar operations for reading and writing various architectural registers and operations that synchronize the clusters with the stream controller. Each instruction in the table operates on an *entire* stream every time it is executed. Furthermore, all stream operands originate in the SRF and stream results are stored back to the SRF. Thus, Imagine is a load-store architecture for streams.

Stream-level instructions are issued to Imagine by the host processor. Once on-chip, the stream controller stores each instruction in a scoreboard and issues them to the proper module as soon as its dependencies are satisfied. Hardware resource dependencies, such as memory port conflicts, are resolved at runtime. Data dependencies are encoded statically and sent to Imagine by the host processor along with each instruction. The high-level language the programmer uses to code stream-level functions is *StreamC*. A *stream scheduler* [5] translates the *StreamC* code to a schedule of stream-level instructions and applies high-level code transformations such as software pipelining and strip-mining. The actual *StreamC* code for the FFT application looks roughly like the pseudocode in Figure 1b. Assuming the stream *'in'* is initially in memory and that the kernel microcode for *fft_stage* is initially in the instruction store in the microcontroller, the StreamC is translated into 12 stream instructions (ignoring any scalar instructions that might be necessary).

The first stream-level instruction is a MEMOP to load the input data into the SRF. This is followed by ten KERNEL instructions, and the last instruction is another MEMOP to store the result in bit-reversed order into SDRAM. The first KERNEL operation will not start until the initial MEMOP completes since the input to the KERNEL operation is the output of the MEMOP. Likewise the input to the second KERNEL operation is the output of the first. The assumption that the microcode initially resides in the instruction store is reasonable since the instruction store is large enough to hold the working set of kernels of most applications. For reference, the microcode for the *fft_stage* kernel is 20 VLIW instructions long (the main loop is only 11), while the instruction store in the microcontroller can store 2K VLIW instructions.

3.2. Kernel-level ISA

A KERNEL operation issued by the stream controller starts the execution of a kernel program on the microcontroller. The microcontroller keeps track of the program counter as it broadcasts each VLIW instruction to all eight clusters in a SIMD manner. Each VLIW instruction consists of sev-

Table 2: Imagine stream-level instruction set

KERNEL	*Execute a kernel operation using stream arguments that reside in the SRF*
MEMOP	*Transfer a stream between SDRAM and the SRF (addressing modes include strided, indexed, and bit-reversed)*
NETOP	*Transfer a stream between the SRF and the network*
HOST_TRANSFER	*Transfer a stream between the host processor and the SRF*
LOAD_PGM	*Transfer a kernel program from the SRF to the microcode RAM in the microcontroller*

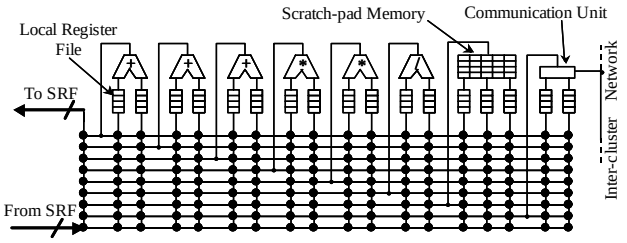


Figure 3: Imagine Arithmetic Cluster Block Diagram

eral ALU operations as well as control signals for every LRF in a cluster. Figure 3 shows the architecture of a single cluster, including the main functional units, the local register files, and the interconnect between the two. Each cluster contains 3 ALUs (which support adds, shifts, and logical operations), 2 multipliers, 1 divide/square root unit, 1 communication unit and 1 local scratch-pad memory. All the arithmetic units support floating point as well as integer arithmetic; the adders and multipliers also support sub-word SIMD arithmetic. The communication unit is an interface to a switch that supports inter-cluster data routing. The scratch-pad memory, a 256×32-bit register file, supports displacement addressing and stores arrays, values spilled from the LRFs, or other data structures such as hash tables or stacks.

The actual program for the *fft_stage* kernel from Figure 1c is programmed in *KernelC*. *KernelC* is a subset of C that abstracts away many details of the internals of the cluster architecture. A compiler that uses *communication scheduling* [7] is used to map a *KernelC* program to a sequence of VLIW instructions. The scheduler applies common optimizations, such as copy propagation and dead code elimination, as well as transformations such as loop unrolling and modulo software pipelining.

4. Discussion

4.1. Micro-architecture

The micro-architectures of the modules on Imagine are designed to be scalable. This is achieved by taking advantage of the properties of the stream model. In particular, Imagine capitalizes on the data-parallel organization of records within a stream, the sequential ordering of accesses to streams, locality of kernel data, and simple control flow within kernels.

Imagine exploits the data-level parallelism (DLP) in streams by executing a kernel on eight successive stream elements in parallel (one on each cluster). The SIMD organization helps Imagine provide the instruction and data bandwidth necessary to operate 48 ALUs. Since every cluster is executing the same VLIW instruction, minimal additional instruction decode and issue logic is required for additional clusters. Furthermore, by partitioning the ALUs into SIMD clusters, the stream register file and local register files can be partitioned as well. This reduces the number of ports into the register files by a factor equal to the number of partitions. Rixner et al. present quantitative studies on the scalability of SIMD partitioned register files [11]. Further SIMD parallelism is available on Imagine since most integer operations have a subword parallel form that operates on two 16-bit values that are packed into one 32-bit register.

The stream model ensures that kernel programs will never access main memory directly. This is because all external kernel inputs and outputs are communicated through data streams, which reside in the SRF. Since the kernel-level instruction set does not contain any other variable latency operations, an accurate static VLIW schedule can be generated. VLIW control is efficient since additional hardware for register renaming, dependency detection, or operation reordering is not necessary.

The SRF takes advantage of the sequential access pattern of streams in order to efficiently support 22 logical ports: 8 cluster (8 words wide), 4 memory, 8 network (2 words wide), 1 host, and 1 microcontroller). The sequential access pattern allows the SRF RAM array to transfer 32 consecutive words of data on every read or write. Since clients access the SRF with requests that are much smaller than 32 words, stream buffers are used to store the intermediate data. This virtualizes access to the SRF, so that one physical port into the SRF RAM can provide enough bandwidth to satisfy the requests from all 22 logical ports. Note that the SRF can satisfy only the *average* bandwidth demand from all the clients. Thus, another advantage of the stream buffers is that they can sustain transient periods of activity that is higher than the peak bandwidth out of the SRF RAM.

One other optimization Imagine employs is partitioning

the local register files in each cluster so that there is one two-ported register file per ALU input. A single crossbar connects each ALU output to every register file input. Rixner et al. show that this register file structure scales with increasing numbers of ALUs better than a single monolithic register file structure [11].

While many media applications adhere to the properties of the stream model listed in this section, there are always practical situations in which code is not perfectly data-parallel or in which data is accessed in a non-regular fashion. For this purpose, Imagine provides communication mechanisms that allow data-parallel partitions to exchange data. In particular, a general permutation instruction shuffles portions of a 32-bit integer in order to communicate between sub-word SIMD partitions. Also, a programmable inter-cluster switch is used to exchange data between clusters and to implement reduction operations. Note that the synchronization in both cases is essentially free since all partitions are operating in lock-step. The last mechanism is a scatter/gather style memory transfer that also allows communication between SIMD partitions, as well as arbitrary data reordering.

The final micro-architectural consideration is conditional execution. While not prevalent in media applications, conditionals do arise and must be dealt with efficiently. A hardware select operation allows execution of data-dependent sections of code using predication. However, predication causes the duty factor of the ALUs to decrease exponentially with the depth of nested if-then-else clauses. Furthermore, load-imbalance can further reduce the duty factor of the ALUs when each cluster has to perform a varying amount of work. For this reason Imagine provides another mechanism, *conditional streams*, that executes data-dependent code more efficiently [4]. Conditional streams efficiently support case-statements, stream combine operations, and operations that require load-balancing across the clusters.

4.2. Bandwidth Hierarchy

The previous section discussed how Imagine takes advantage of the stream model in order to provide a scalable micro-architecture. This allows Imagine to provide 48 ALUs on a single chip. This section will focus on how Imagine provides the necessary bandwidth to support 48 ALUs by taking advantage of the *locality* exposed by the stream model. This is necessary because it is impractical to provide data on every cycle to this many ALUs using off-chip DRAM or even a single monolithic register file. Imagine's bandwidth hierarchy, however, increases the available bandwidth by an order of magnitude at each level of the hierarchy by taking advantage of the locality exposed by the stream programming model. The bandwidth hierarchy is integral to scaling this architecture to support more than a few ALUs.

In the stream model, the greatest data bandwidth demand is within a kernel. Since local routing resources are plentiful, Imagine economically provides the most data bandwidth within a cluster, and maps kernel execution onto the clusters. Also recall that the stream model exposes the producer-consumer locality of streams. Thus, Imagine maps stream communication onto the second level of its hierarchy, the SRF. The SRF captures the locality of temporary streams, such as those between successive calls to the *fft_stage* kernel. This limits the amount of data that is transferred between off-chip SDRAM and Imagine to only the data which is truly global, such as the initial input and final output data in an application. The actual bandwidth numbers for different communication paths on Imagine are summarized in Table 3. The lines in bold correspond the bandwidth levels discussed in this section. Note that the ratio of bandwidths provided by Imagine is 1:12:207. Also, other references provide details on how well media application demands map to this hierarchy provided by Imagine [10, 6].

Table 3: Maximum rates of the Imagine Processor

	Per Cluster (W/cycle)	Aggregate (GB/s)
Intra-cluster BW	**34**	**435**
Peak SRF BW	8	140
Average SRF BW	**2**	**25.6**
Inter-cluster BW	1	12.8
Network BW	0.5	6.4
DRAM BW	**0.16**	**2.1**

5. Case Studies

This section will investigate how the performance of real programs scale on Imagine. One indicator of scalability is the speedup of programs on two machines that differ only in the number of clusters: the first machine has only 1 cluster, the second machine has 8, like Imagine. In order to achieve a high speedup from the 1-cluster machine to the 8-cluster machine, the application has to contain DLP to begin with, the stream model must expose it, and Imagine must be able to exploit it. This aspect of scaling (data-parallelism across the clusters) will be explored in order to better understand how real programs scale. Note, however, that the speedups presented here do not take into account VLSI implementation details since they are outside of the scope of this paper.

Table 4 lists the speedup realized for one kernel and two applications. The *fft_stage* kernel is similar to the one discussed in an earlier section, except it uses a radix-4 algorithm. The *Polygon Rendering* application is the ADVS-1 benchmark presented in [8]. The *Depth Extraction* application is the same as the one in Table 1.

The *fft_stage* kernel was chosen because it is a good ex-

Table 4: Speedup of 8 clusters over 1 cluster

Name (kernels in italics)	Kernel Speedup	Application Speedup
radix-4 fft_stage	6.5	–
Polygon Rendering	5.3	4.6
Depth Extraction	6.2	5.7

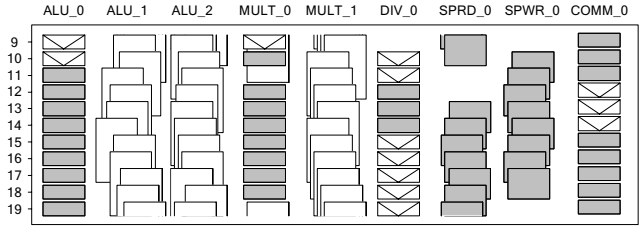


Figure 4: Radix-4 FFT kernel schedule

ample of a kernel that is not perfectly data-parallel, and hence incurs some overhead when implemented on Imagine's SIMD clusters. Figure 4 helps illustrate the type and number of operations that are required due to this overhead. This figure shows a visualization of the inner loop of the radix-4 FFT kernel after scheduling (and software pipelining). This schedule was created using *iscd*, the KernelC scheduler described in Section 3.2. The vertical axis in the figure is instruction number (time moves downwards, the height of a box is the latency of that operation); each column corresponds to a different functional unit. The white unmarked operations are floating point operations to compute the numerical FFT results. The eight gray operations on the COMM_0 unit are inter-cluster communication operations which transfer data between cluster partitions. The operations on the SPRD_0 and SPWR_0 units are indexed reads and writes to the scratch-pad.[1] The other gray boxes are all hardware select operations.

All the gray operations are not present in the 1-cluster version, and account for the sub-linear speedup of the kernel. They are needed to transpose the data among the cluster partitions, so that the data is reordered according to the FFT algorithm. The selects and scratch-pad operations are necessary to allow each cluster to choose a different value to communicate or output — they are essentially simple conditionals. There are a total of 20 selects, 16 scratch-pad operations, and 8 inter-cluster communications — all of which are unnecessary on the 1 cluster machine.[2] On the other hand, notice that no extra operations are required for synchronization between the clusters since the clusters are synchronized by definition in the SIMD execution model. The end result is that despite the extra operations, the *fft_stage* kernel still achieves a speedup of 6.5.

An interesting note is that the scheduler exploited the available ILP in the cluster. In fact, the inner loop of this kernel achieves 92% of the peak instruction rate in the clusters. The visualization also shows the amount of overhead in this kernel due to software pipelining and the distributed register file organization. All the *envelope-style* operations were added during the scheduling process in order to delay a value for a later software pipeline iteration, to move a value from one local register file to another, or to do both.

The two applications shown in Table 4, *Polygon Rendering* and *Depth Extraction*, provide more examples of how kernel performance scales from the 1-cluster to 8-cluster machine. The speedups of just the kernels in the application are 5.3 and 6.2 respectively. These speedups were measured using the total kernel runtime for each application on each machine. The runtimes were measured on *isim*, a cycle-accurate stream processor simulator. The deviation from the ideal speedup of 8.0 occurs for similar reasons as for the *fft_stage* kernel. In addition, the *Polygon Rendering* application contains several large conditional blocks. These were mapped to conditional streams to retain a high duty factor for the ALUs as well as to improve load-balance among the SIMD partitions.

Kernel speedup is not the entire story for these two applications, however. That is because any overhead in the application, such as priming and draining execution pipelines, becomes a larger portion of the runtime on the larger machine. Furthermore, SRF bandwidth and DRAM bandwidth were kept constant for these two machines. So the 8-cluster machine has less SRF and DRAM bandwidth per ALU, and therefore is stalled waiting for results from DRAM more often. The end result is that the overall application speedup of the *Polygon Rendering* application is 4.6 and for the *Depth Extraction* application it is 5.7. Thus, for the particular configuration of the Imagine processor, these overheads only caused the speedups to deteriorate 8–13% from the pure kernel speedups. These overheads, however, may have a smaller or larger impact on different machine configurations, and will surely have a larger impact as a machine scales to larger numbers of ALUs.

6. Related Approaches

The Imagine stream architecture builds upon several existing ideas. Some of these include vector processors, task-parallel stream processors, VLIW media processors, DSPs, graphics coprocessors, and subword SIMD parallel instruction sets. Only the first two will be discussed here — further comparisons with related approaches can be found in [6].

The vector model is similar to the stream model: vector

[1] The scratch-pad ports went completely unused in the 1-cluster implementation of *fft_stage*.

[2] These operations are not shown in the pseudocode in Figure 1, but would be required in a KernelC implementation of *fft_stage*.

architectures [12] take advantage of the data parallelism between consecutive elements in a vector in the same way a stream processor does. However, vector architectures do not have an analog for the local register files in stream architectures. This increases the demand on vector register file bandwidth and ultimately limits the scalability of a vector architecture. Furthermore, a stream processor performs all the operations required for a record before processing the next record. This creates a relatively small working set of temporary values. A vector processor instead performs the first operation on all the elements in a vector, and then the next operation on all the elements in the vector. This order of execution creates a larger set of temporaries, increasing the demand on the vector register file capacity.

Another related approach is to exploit the task-level parallelism (TLP) exposed by the stream model. Programmable architectures in this class [14, 1, 13] execute multiple kernels concurrently and take advantage of the producer-consumer locality and TLP inherent in stream programs by passing streams directly between kernels. These architectures can scale to larger numbers of ALUs by either adding more ALUs to each kernel execution unit or by adding more kernel execution units. The work presented in this paper concentrates solely on scaling the performance of a *single* kernel as much as possible by taking advantage of the data- and instruction-level parallelism (DLP and ILP) in the stream model. It is an open research problem to quantify the performance of applications on a spectrum of architectures, each with a different amount of scaling in the DLP, ILP, and TLP axes.

7. Conclusions

The *Imagine Stream Processor* supports 48 ALUs on a single-chip, and sustains up to 15 GOPS on realistic media applications such as MPEG-2 encoding. The architecture of the chip is based on the concept of streams. The stream model exposes the parallelism and locality in an application. Imagine provides a bandwidth hierarchy to take advantage of the locality exposed by the stream model, and provides over two orders of magnitude more bandwidth at the ports of the ALUs than available to off-chip SDRAM. Furthermore, the architecture can be efficiently scaled: the Imagine prototype supports 48 parallel ALUs on a single chip. Finally, stream applications map well to this hierarchy enabling Imagine to operate all 48 ALUs at a high duty factor.

Future work will look at scaling the performance beyond a single node. The expectation is that the stream discipline will allow multiple-node systems to take advantage of the parallelism in applications in the same way as a single node does. Furthermore, the stream model may make automatic domain decomposition to systems with very large numbers of nodes a tractable problem. One area of current research is to attempt to solve some of these issues in order to apply the concepts of the Imagine architecture to systems with thousands of nodes that are executing large scale scientific codes [2]. A large system like this also requires innovative language design and compilation techniques. Imagine's *KernelC* and *StreamC* are first steps in the right direction, but they are somewhat specific to media processing and to the Imagine architecture. The design of a high-level stream language that can be applied to a more general class of domains and architectures is an immediate research objective.

References

[1] E. Caspi, A. Dehon, and J. Wawrzynek. A Streaming Multi-threaded Model. In *Proceedings of the Third Workshop on Media and Stream Processors*, pages 21–28, Austin, TX, Dec 2001.

[2] W. J. Dally, P. Hanrahan, and R. Fedkiw. A Streaming Supercomputer. Stanford Computer Systems Laboratory White Paper, September 2001.

[3] 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, June 1996.

[4] U. J. Kapasi, W. J. Dally, S. Rixner, P. R. Mattson, J. D. Owens, and B. Khailany. Efficient Conditional Operations for Data-parallel Architectures. In *Proceedings of the 33rd IEEE/ACM International Symposium on Microarchitecture*, pages 159–170, December 2000.

[5] 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.

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

[7] P. Mattson, W. J. Dally, S. Rixner, U. J. Kapasi, and J. D. Owens. Communication Scheduling. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, November 2000.

[8] J. D. Owens, W. J. Dally, U. J. Kapasi, S. Rixner, P. Mattson, and B. Mowery. Polygon Rendering on a Stream Architecture. In *Proceedings of the 2000 SIGGRAPH / Eurographics Workshop on Graphics Hardware*, pages 23–32, August 2000.

[9] M. C. Pease. An Adaptation of the Fast Fourier Transform for Parallel Processing. *Journal of the ACM*, 15(2):252–264, April 1968.

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

[11] S. Rixner, W. J. Dally, B. Khailany, P. R. Mattson, U. J. Kapasi, and J. D. Owens. Register Organization for Media Processing. In *Proceedings of the 6th International Symposium on High-Performance Computer Architecture*, pages 375–386, January 2000.

[12] R. Russell. The Cray-1 Computer System. *Comm. ACM*, 21(1):63–72, Jan 1978.

[13] M. B. Taylor and et al. The Raw Microprocessor: A Computational Fabric for Software Circuits and General Purpose Programs. *IEEE Micro*, Mar/Apr 2002.

[14] J. V. M. Bove and J. A. Watlington. Cheops: A Reconfigurable Dataflow System for Video Processing. *IEEE Trans. on Circuits and Systems for Video Tech.*, pages 140–149, April 5 1995.