Stanford University Concurrent VLSI Architecture Memo 122

Stanford University
Computer Systems Laboratory

Stream Scheduling

Ujval J. Kapasi, Peter Mattson, William J. Dally, John D. Owens, Brian Towles

Abstract

Media applications can be cast as a set of computation kernels operating on sequential data streams. This representation exposes the structure of an application to the hardware as well as to the compiler tools. *Stream scheduling* is a compiler technology that applies knowledge of the high-level structure of a program in order to target stream hardware as well as to perform high-level optimizations on the application. The basic methods used by stream scheduling are discussed in this paper. In addition, results on the Imagine stream processor are presented. These show that stream programs executed using an automated stream scheduler require on average 98% of the execution time that hand-optimized assembly versions of the same programs require.

Keywords: stream scheduling, strip-mining, software-pipelining, stream allocation.

*This report also appears in the *Proceedings of the 3rd Workshop on Media and Stream Processors*, held December 2, 2001 in Austin, TX, in conjuction with the 34th International Symposium on Microarchitecture (MICRO-34).

1 Introduction

Media applications can be cast as a set of computation kernels operating on sequential data streams. This method of representation, the *stream programming model*, exposes the high-level structure of a program to software tools and to the hardware. Specifically, the inherent locality and concurrency present in an application are exposed by the stream model. For example, consider the sample application in Figure 1. The top half of the figure shows an abstract flow graph representation using *kernels* and *streams*. There are three computation kernels, *Op1*, *Op2*, and *Op3*, two input streams, *a* and *b*, two intermediate streams, *c* and *d*, and one output stream, *e*. The bottom half shows the same example written in *StreamC*, which is a C++ based language used to express streaming applications. Note that the *StreamC* code has retained all the high-level structural information of the flow-graph. Using this information is the key to building efficient hardware and optimizing application performance.

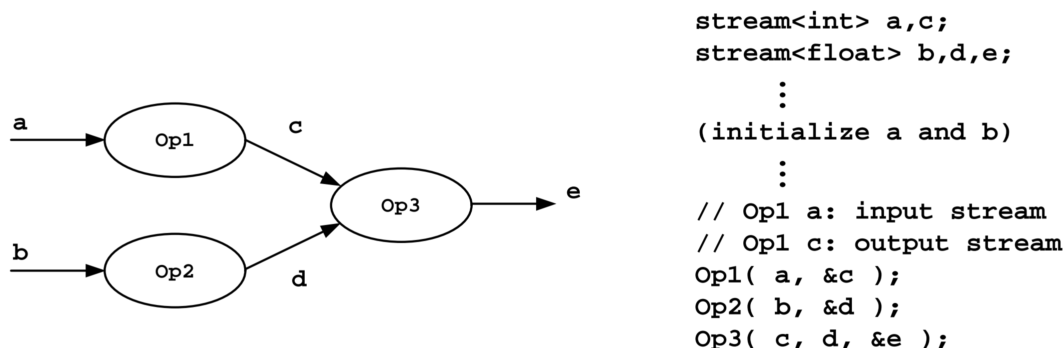


Figure 1: Sample application flowchart and *StreamC* code

The stream programming model can be efficiently implemented in hardware because its requirements are well-matched to the capabilities of modern VLSI [1]. The Imagine stream architecture is an example of a direct mapping of the stream programming model to hardware. While this architecture is efficient from a VLSI perspective, it presents new challenges to the programming tools. Operations on streams must be coordinated to maximize concurrency and to efficiently utilize the on-chip stream storage. Off-chip bandwidth must be used only when necessary, since spilling a stream to memory could require reading and writing thousands of extra data elements. Furthermore, data dependence analysis must be performed on entire stream operations instead of individual scalar operations. Finally, streams that are too large to fit completely on-chip must be dealt with in a reasonable fashion.

While a stream compiler must address these challenges, it can also take advantage of the structure present in stream programs to perform high-level optimizations. Task-level parallelism, for example, can be easily extracted from this representation of the application. In the example we can see that the kernels *Op1* and *Op2* can be scheduled to execute in parallel. A stream compiler can also take advantage of producer-consumer locality of intermediate streams, since this is also exposed by the programming model. For example, with a software controlled on-chip memory, streams *c* and *d* never need to be transferred to off-chip memory. This can reduce the number of memory accesses by a factor of 3–11 for typical applications.

Stream scheduling is the technique used by a compiler to leverage knowledge of the structure of a program exposed by the streaming model in order to address new challenges posed by stream hardware as well as to perform high-level optimizations on a stream program. While previous systems have been able to extract high-level information in order to simultaneously compute and perform stream memory operations [2, 3], they use a set of FIFO queues for stream routing instead of a general stream register file. This paper presents an overview of scheduling for a stream architecture, and is explored in detail by Mattson [4]. The next section, Section 2, presents a basic streaming system framework within which stream scheduling can be applied. Section 3 discusses stream scheduling in detail. Performance results are presented in Section 4. The current status of this research is presented in Section 5 along with future research directions, and conclusions are drawn in Section 6.

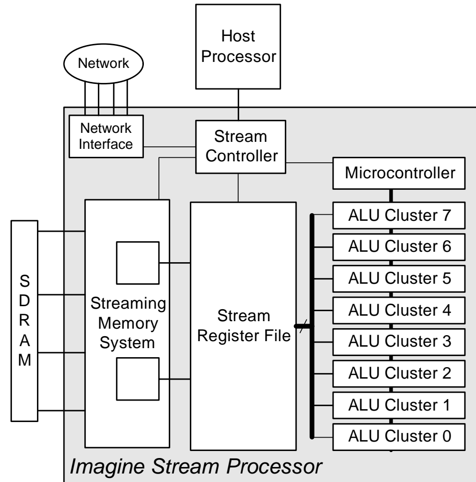


Figure 2: Imagine architecture block diagram

2 A Stream System

The ultimate goal of stream scheduling is to minimize the execution time of a stream program. A stream program describes two different levels of execution, namely the stream and kernel levels. Code written for the kernel level is expressed in *KernelC*. At the stream level, code is expressed in *StreamC*. The two levels are executed by separate hardware on a stream architecture. An example of a stream architecture is the Imagine processor, shown in Figure 2. This architecture and its corresponding programming system (*StreamC* and *KernelC*) will be used as the demonstration platform throughout this paper.

In the Imagine architecture, streams are routed globally on-chip via a software-controlled Stream Register File (SRF) that contains 128KB of storage. The eight arithmetic clusters, which efficiently support a total of 48 arithmetic units, execute computation kernels in a SIMD fashion. Off-chip DRAM is accessed through a streaming memory system [5] optimized for throughput. Applications are written in *StreamC* and *KernelC* and eventually compiled to native Imagine instructions. *KernelC* functions are stored in an on-chip memory in the microcontroller. The microcontroller sequences the individual *KernelC* instructions and broadcasts them to the SIMD arithmetic clusters.

The following walk-through will illustrate how an application actually executes on this architecture and how stream scheduling fits into the whole system. The *StreamC* code for an application is compiled into a series of stream operations. A stream operation is applied to an entire stream. For example, one stream operation might load an entire stream from memory into the SRF. A stream operation might also execute an entire kernel that reads one or more streams from the SRF and outputs one or more streams back to the SRF. Stream scheduling is responsible for the choice of operations for a particular *StreamC* program, their relative order, and the location and length of all streams. As will be discussed, controlling these parameters can reduce overall execution time of a program.

Once the stream operations have been generated, the host processor issues the operations to the stream controller on the Imagine chip. The host processor also sends pre-computed dependency information to the stream controller, which stores it in a scoreboard. The scoreboard determines when all of an operation's dependencies have been satisfied, and at that point the stream controller can issue the operation to the relevant module.

For example, consider a kernel operation that uses one input stream and one output stream, and a memory operation that stores the output stream to main memory. Assume both operations initially reside in the stream controller. The scoreboard determines when the kernel's input stream is ready in the SRF and when the SRF locations intended for its output stream no longer contain live data. At that point, the stream controller will issue the kernel operation and the kernel will be executed by the clusters. At the end of the kernel execution, the scoreboard will signal the stream controller to issue the memory operation. Once the final word of the stream is stored to off-chip memory, the scoreboard will inform the host-processor that this program has completed. Note that only two stream operations were required to process and store a whole stream of input data. Further note that a more complex example can actually have concurrent execution of a kernel and two independent memory stream transfers.

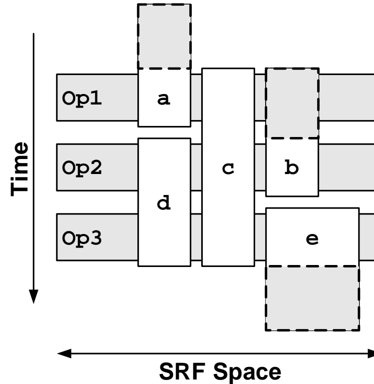


Figure 3: Allocation of buffers in the SRF

3 Scheduling

Stream scheduling coordinates both the execution of kernels and the movement of streams in order to minimize program execution time. To meet this goal, stream scheduling addresses three key areas: (1) efficient allocation of the on-chip stream register file (SRF), (2) exploiting producer-consumer locality between kernels in order to reduce off-chip bandwidth, and (3) maximizing concurrency. Since an application is expressed using the stream programming model, a compiler can analyze the high-level structure of the application in order to address all three issues. The methods of doing this are discussed in the following three subsections.

3.1 Allocation

The stream scheduler builds the dependency graph of the application based on the streams that each operation accesses. For example, the call to *Op1* in Figure 1 contains two *stream accesses*, whereas the call to *Op3* contains three *stream accesses*. The scheduler assigns a *buffer* in the SRF to each stream access. Where possible, the same buffer is assigned to every access of a particular stream. For example, the buffer for stream *c* is assigned to both the access for *Op1* and the access for *Op3*.

Each buffer is simply a set of logically contiguous locations in the SRF that will store a stream for a continuous period of time. The size of each buffer is determined by the length of the corresponding stream. The amount of time for which a buffer must allocate its space in the SRF is determined by the lifetime of the stream it is holding. The overall allocation problem requires placing the buffers in the SRF such that the amount of data in the SRF is never greater than the size of the SRF at any point in time. This problem is similar to register allocation, except that buffers have the additional dimension of size.

For the example presented earlier, Figure 3 shows an efficient allocation for the buffers for the five streams. In this diagram, the vertical dimension represents time. Operations scheduled earlier appear higher in the diagram, so *Op1* is scheduled before *Op2*. The horizontal dimension represents SRF space and the width of a stream represents its length. For example, stream *e* contains more elements than does stream *b*. This diagram also shows the lifetime the stream assigned to each buffer. For instance, the figure clearly shows that stream *c* is generated by *Op1* and is unneeded after *Op3*.

The allocation shown in the figure is for this particular ordering of operations. If *Op2* were listed first, the allocation would differ. Further, note that the diagram above does not contain any memory operations. For this example, assume that the streams *a* and *b* need to be loaded from off-chip memory, and that the stream *e* needs to be stored to off-chip memory. In this case, we would need to execute a stream memory load operation before kernel *Op1* for stream *a*. This would require extending the the buffer for *a* to an earlier point in time. This is indicated in the diagram by the dotted gray boxes. Note that the memory operation for stream *b* can execute in parallel with the execution of kernel *Op1* on the Imagine architecture.

One common situation which must be considered is when a stream is too large to completely fit within the SRF. In this case the stream must be *double-buffered*. First a portion of the stream is loaded into a buffer in the SRF. Then, while a kernel is reading that portion of the stream, the next part of the stream is loaded into a different buffer in the

SRF. After the kernel exhausts the elements in the first buffer, it starts reading from the second buffer while the third part of the stream is now loaded into the first buffer.[1] In this manner streams of arbitrary lengths can be handled. Furthermore, on the Imagine architecture, the arithmetic clusters treat double-buffered and regular streams in a similar fashion. Thus, no extra kernel code is required to distinguish between one long double-buffered stream and a shorter stream that fits in the SRF.

Most applications are more complex than the example in Figure 1 — they usually execute many more stream operations and use streams of varied size and lifetime. One possible algorithm to deal with more complex applications is presented below. This algorithm tries to reduce the amount of spilling required to off-chip memory and tries to increase the amount of possible concurrency. In Section 4 its performance will be compared to a simpler allocation algorithm.

1. *Determine which stream accesses are double-buffered*

   Guided by a set of heuristics, the scheduler chooses which streams to double-buffer and sets the buffer sizes.

2. *Assign stream accesses to buffers*

   The stream scheduler attempts to assign accesses to buffers based on architectural and program constraints. It tries to maximize the number of accesses that share a single buffer when making assignments. The initial size and initial lifetime of each buffer is set based on the stream accesses assigned to it.

3. *Mark stream accesses that require memory accesses*

   Using dataflow analysis, the stream scheduler determines where memory accesses are required to synchronize data between buffers. For example, if one buffer contains data that will be required later by a buffer in a different location in the SRF, a memory transfer is required to propogate the changes from the first buffer to the second.

4. *Repeatedly divide each buffer if it is possible to do so without requiring additional memory accesses*

   A buffer is divided if the accesses assigned to it can be divided into two disjoint groups without inducing extra memory accesses. This is useful because smaller buffers allow more flexibility during the packing step.

5. *Extend buffers to account for memory operations*

   These extensions correspond the dotted gray boxes in Figure 3. This step simply assumes that as many memory operations as necessary can execute in parallel with a kernel operation.

6. *Position buffers in the SRF*

   Even the easy case for this allocation problem — which is when all streams are of unit length — reduces to a $\mathcal{NP}$-complete graph coloring problem just like scalar register allocation. Thus, finding a scheme which packs all the buffers into the SRF without spilling requires a set of heuristics.

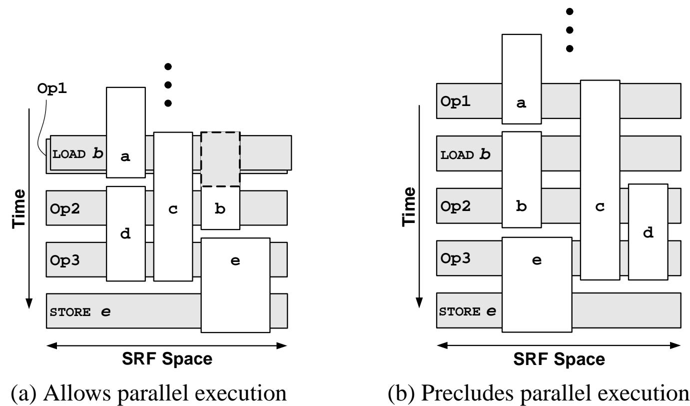
7. *If the buffers do not fit in the SRF, reduce a buffer and repeat steps 3-7*

   If the packing was unsuccessful, the scheduler uses a set of heuristics to choose a buffer to reduce. If the buffer chosen for reduction is used for double buffering a stream, its size can be reduced. Otherwise, the buffer is reduced by splitting its lifetime into two smaller lifetimes and spilling to memory in between.

3.2 Locality

Where possible, the stream scheduler takes advantage of the producer-consumer locality between kernels. All accesses to a particular stream are assigned to the same buffer so that no spilling is required. For example, stream *c* in the example never needs to be stored to off-chip memory. If, however, all the accesses cannot be assigned to the same

[1] Stream accesses on Imagine are atomic for reading and writing. If a stream is being written into it cannot be read from until the writing operation has finished.



(a) Allows parallel execution (b) Precludes parallel execution

Figure 4: Accounting for parallel execution of *Op1* and LOAD *b*

buffer, the stream must be stored to memory from one buffer, and read in later to a different buffer before being used by a following stream operation.

An important optimization that takes advantage of producer-consumer stream locality is *strip-mining*. Strip-mining segments a large input stream into smaller batches so that all intermediate streams produced while processing a batch fit in the SRF [6]. Since many applications operate on streams that are larger than the SRF, this optimization is essential for good performance. Without strip-mining, all temporary results would have to be stored back to memory and reloaded using double-buffering.

The *strip-size* can be defined as the size of the initial input stream(s). The strip size determines the amount of data in the intermediate streams. A large strip-size is desirable to mitigate kernel startup and shutdown overheads, especially when kernels are heavily software pipelined. However, larger strip-sizes increase the chances that an intermediate stream will have to be spilled to off-chip memory in order to succeed SRF allocation. A reasonable point on this trade-off curve is to choose a strip-size as large as possible without requiring any intermediate streams to be spilled.

3.3 Parallelism

Based on the particular stream architecture being targeted, stream operations can be generated in a way that takes advantage of the task-level parallelism in a program. For example, on Imagine, two streaming memory operations and one kernel may be executed in parallel, as long as there are no data dependencies between them. In the example from Figure 1, for instance, the memory load for *b* could execute in parallel with the kernel *Op1*. However, this must be taken into account when allocating the SRF. Figure 4a shows the allocation of the SRF when the possibility of parallel execution is taken into account (assume that the stream *a* has been generated by another kernel). On the other hand, if the operations are simply ordered arbitrarily, an allocation such as that of Figure 4b might result. Note that in this case the LOAD *b* instruction cannot execute in parallel with *Op1* because of a false dependency.[2] Thus, the amount of concurrency a program achieves can be increased with careful SRF allocation.

Another issue that can affect the performance of a stream program is the execution time of each operation. The actual execution time for a stream operation can be dependent on conditions that differ each time the instruction is issued. For example, the memory system can service two independent memory stream transfers in parallel. Since SDRAM access latencies are dependent on the state the SDRAM is in, accesses from one stream can leave the SDRAM in a state that can affect the latency of the accesses from the other stream. However, variation in operation instruction time can be dealt with by the on-chip stream controller since it can reorder operations dynamically.

The stream controller, however, has a limited scoreboard size and cannot perform global reordering optimizations. Instead, stream scheduling can capitalize on its knowledge of program structure to perform global operation reordering. One effective global optimization is software pipelining of stream operations [7]. Software pipelining can reorder operations in order to increase the achieved parallelism between memory transfers and kernel execution. Software pipelining does, however, come at the cost of increased SRF demand since more buffers are usually live at any particular point in time.

[2]Note that a scalar processor could handle this case with hardware register renaming. This functionality is harder to provide in stream hardware since streams of different sizes can be allocated at any arbitrary location in the on-chip memory.

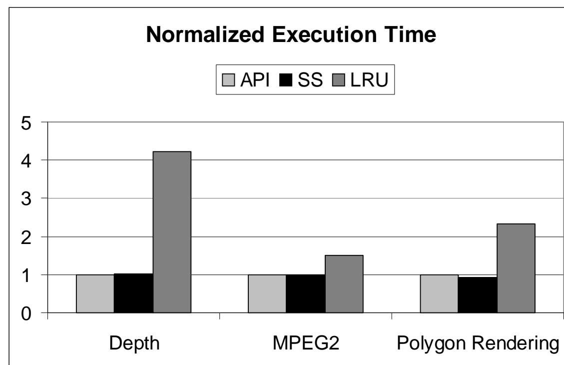


Figure 5: Benchmark Runtimes

4 Results

The Imagine architecture and programming system serves as the platform for obtaining results. A cycle-accurate simulator was used to collect final performance numbers. Furthermore, due to specific peculiarities of the Imagine architecture, there are some differences in the scheduling technique used by the actual stream compiler from the conceptual scheduling presented in Section 3.

The main difference is that the actual tool compiles each basic block on the fly in an interpretive framework. Instead of statically compiling the stream program, the interpreter schedules every basic block the first time that block executes. During this first scheduling phase, it uses a simple but inefficient SRF allocation scheme. During the first execution of the basic block, the interpreter also records relevant information about each stream operation. Once the interpreter completes execution of the basic block, this information is used to schedule the operations using stream scheduling. It is at this point that the streams are allocated in the SRF and that software pipelining can be applied.[3] Every subsequent execution of the basic block will use the optimized schedule generated by the stream scheduler. These precomputed basic block schedules can also be saved to disk and used for future executions of the application.

The other difference is that strip-mining is not automated. Some programs can not be strip-mined by arbitrarily splitting the input stream into smaller batches. This is because kernels often have state carried between stream elements and cannot be stopped and restarted without destroying their state. For this reason, the stream scheduler instead performs an analysis to determine the ideal strip size required to maximize SRF usage without overflowing, and provides this information as feedback to the programmer. At that point, the programmer can manually strip-mine the loop using the optimal strip-size.

In order to evaluate the performance of the stream scheduler, benchmarks compiled using the stream scheduler were compared to a control version that was written in a low-level API and optimized by hand. Furthermore, to evaluate the SRF allocation algorithm presented in Section 3.1, it is compared to a simpler algorithm that allocates SRF space via a least recently used (LRU) replacement scheme. These three versions are referred to as SS, API, and LRU respectively. Three applications were tested: a stereo depth extractor [8], MPEG-2 encode, and a polygon rendering pipeline [9].

Figure 5 shows the runtime of these three applications, normalized to the data for API. Comparing the results for API and SS we can see that the automated stream scheduler performed close to, or better than, the hand-optimized assembly (98% better on average). The difference in execution time between these two versions for the DEPTH and MPEG2 benchmarks were both within 3%. However, note that the stream scheduled version outperformed the hand coded version by 10% on the POLYGON RENDERING benchmark. This is mainly because stream scheduler was able to generate a more optimal layout for the SRF. Since the application is strip-mined, a more optimal layout can use a larger batch size without spilling to main memory. This improves performance because a larger batch size decreases the kernel startup and shutdown overhead incurred when executing the entire application. In this case, SS successfully allocated the SRF without spilling using a batch size of 256 input triangles whereas the API SRF allocation could only support 80 input triangles per batch without spilling.

Further, note that SS executed the applications an average of 43% faster than LRU. This performance difference can

[3] The basic block schedule can also be generated offline by executing the application on a functional simulator.

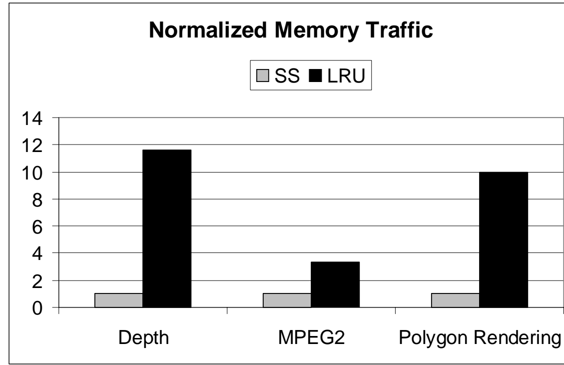


Figure 6: Memory Traffic for SS and LRU

be attributed to the amount of memory traffic generated. Figure 6 shows a graph containing the amount of generated memory traffic for each application, normalized to the SS data. SS used global knowledge of the program to allocate the SRF in a manner that would avoid memory spills where possible. LRU, which used the same batch size as SS, did not take advantage of the same global high-level information. In particular, it based its decisions solely on past stream operations, and did not make decisions to accommodate future operations. As a result, it generated less optimal SRF allocations that required frequent memory spilling.

5 Status and Future Work

Currently we have implemented the three applications listed above, as well as several variations of the basic graphics rendering pipeline. All the results given were obtained using a fully automated software tool suite. The tool suite can be configured via a machine description file to target a range of stream architectures. The kernels were compiled with a VLIW scheduler developed in-house to target the arithmetic clusters. All stream-level code was scheduled using the automated StreamC compiler described in Section 5.

In addition to the results obtained from a cycle-accurate simulator, we plan to test the performance of these applications on actual hardware, as Imagine chips are expected to arrive during Q1 2002. We hope to distribute 2-node Imagine systems to various parties interested in developing software for Imagine. This will provide a great amount of feedback on usability and performance that we can use to fine-tune and extend the capabilities of the stream scheduler.

The stream scheduler is currently a basic block compiler implemented within the framework of an interpreter. A more powerful and robust tool would statically analyze the whole application, including all *StreamC* and *KernelC* code, and would be able to generate a potentially more optimal profile. This is something that we plan to look at in the future.

Finally, the stream programming model maps easily to multiprocessor systems. Streams can be communicated between nodes using the network interface, which transfers whole streams via an interconnection network. Kernels can be pipelined across nodes to exploit task-level parallelism, or the same kernel can operate on different portions of a stream on different nodes in order to exploit data-level parallelism. Automating this mapping and automating the control of such a system is an open problem that we plan to address in the future.

6 Conclusions

The stream programming model exposes high-level information about the concurrency and locality in a program. Hardware to execute applications written in this model can be implemented very efficiently in VLSI. Compiling stream programs to this hardware presents several new challenges as well as opportunities. Any stream scheduling technique must address these new challenges. At the same time, however, it can take advantage of the stream programming model in order to perform high-level optimizations not possible otherwise.

This paper presents these new challenges as well as techniques for handling them. On-chip memory allocation, off-chip bandwidth management, as well as inter-stream dependency analysis are examples. Moreover, new optimizations are presented that are made possible by the stream programming model. In particular, taking advantage of producer-consumer locality as well as task-level parallelism can reduce off-chip bandwidth and increase performance. Stream scheduling deals with all of these, and can be applied to a variety of target architectures. The particular framework presented here incorporated stream scheduling into a compiler for the *StreamC* language that targeted the Imagine architecture. Compared to hand-optimized assembly versions of three media processing application benchmarks, the stream scheduled versions ran in 98% of the execution time. Thus, the fully software optimized versions ran close to, if not faster, than the same application written in hand-optimized assembly.

References

[1] Scott Rixner, William J. Dally, Ujval J. Kapasi, Brucek Khailany, Abelardo Lopez-Lagunas, Peter R. Mattson, and John D. Owens, “A bandwidth-efficient architecture for media processing,” in *International Symposium on Microarchitecture*, 1998, pp. 3–13.

[2] M. E. Benitez and J. W. Davidson, “Code generation for streaming: an access/execute mechanism,” in *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating System (ASPLOS)*, 1991, pp. 132–141.

[3] S. McKee, C. Oliver, W. Wulf, K. Wright, and J. Aylor, “Design and evaluation of dynamic access ordering hardware,” in *Proc. 10th ACM International Conference on Supercomputing*, May 1996.

[4] Peter Mattson, *Programming System for the Imagine Media Processor*, Ph.D. thesis, Stanford University, 2002.

[5] Scott Rixner, William J. Dally, Ujval J. Kapasi, Peter R. Mattson, and John D. Owens, “Memory access scheduling,” in *ISCA*, 2000, pp. 128–138.

[6] M. Wolfe, “More iteration space tiling,” in *Proceedings of the Supercomputing 89*, New York, NY, 1989, pp. 655–664, ACM Press.

[7] M. Lam, “Software pipelining: An effective scheduling technique for VLIW machines,” in *Proceedings of the ACM SIGPLAN ’88 Conference on Programming Language Design and Implementation (PLDI)*, 1988, vol. 23, pp. 318–328.

[8] Takeo Kanade, Hiroshi Kano, and Shigeru Kimura, “Development of a video-rate stereo machine,” in *Proceedings of the International Robotics and Systems Conference*, August 1995, pp. 95–100.

[9] John D. Owens, William J. Dally, Ujval J. Kapasi, Scott Rixner, Peter Mattson, and Ben Mowery, “Polygon rendering on a stream architecture,” *2000 SIGGRAPH / Eurographics Workshop on Graphics Hardware*, pp. 23–32, August 2000.