Ubiquitous Stream Programming to Facilitate the Migration to Multicore Architectures

Matthew Drake, David Zhang, Michael Gordon, Janis Sermulins, William Thies, Allyn Dimock, Rodric Rabbah, and Saman Amarasinghe
Massachusetts Institute of Technology
Computer Science and Artificial Intelligence Laboratory

Abstract

The goal of the StreamIt project is to become the language of choice for streaming applications, which include multimedia codes, as well as cryptography, networking and security, and some forms of scientific computation. StreamIt provides a high level stream language and a stream-aware compiler that leverages domain specific language constructs to deliver high performance from high levels of abstraction. We believe that StreamIt will lead to a dramatic improvement in programmer productivity, program robustness, portability, and performance scalability on commodity multicore architectures. A key component of the research is to also provide seamless integration with other programming practices, because large applications have various properties that may require different language abstractions. This paper outlines a methodology that allows StreamIt programs to run as part of the X10 virtual machine. X10 is one of three general purpose languages under development from vendors designing large scale parallel platforms.

1. Introduction

Multicore architectures offer a significant amount of coarse-grained parallelism on chip. They also increase the burden on programmers who now have to explicitly extract coarse-grained parallelism from their codes in order to leverage the compute potential available in emerging processors. Multimedia applications are especially challenging because they require stream abstractions that are not easily represented using current programming paradigms, and general purpose compilers eschew stream-aware optimizations. As a result, applications are hand-coded for performance, and this practice precludes portability and obfuscates readability.

The StreamIt project at MIT provides a language [10] with simple abstractions for parallel programming, and a stream-aware compiler [1, 5, 7, 8] that delivers scalable high performance on a variety of commodity processors [5, 9]. StreamIt is focused on stream programming ubiquitous to multimedia and digital signal processing, as well as applications drawn from cryptography, security, networking, and some forms of scientific computing. In StreamIt, computation is carried out by autonomous filters, and data are communicated over well structured topologies that interconnect the filters. The StreamIt programming model allows the programmer to build an application by connecting components together into a stream graph. The programmer is relieved of the burden of explicit buffer management and complex modulo index expressions, and naturally exposes the parallelism and communication patterns inherent to their codes. The end result is clean, malleable, and portable code [3].

The StreamIt project is also concerned with providing seamless integration between streaming codes and other programming practices. This is necessary because large scale applications have various properties, each requiring a different language approach. The alternative is to use a single general purpose language for an applica-

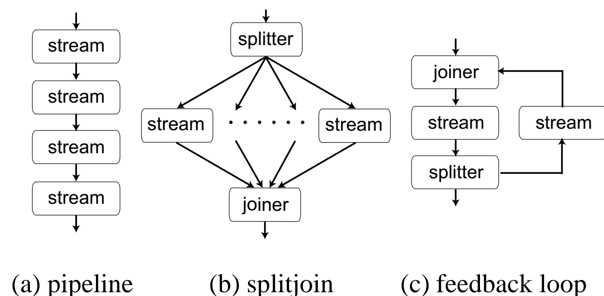
tion, although this practice hides program details that are necessary for good compilation strategies. StreamIt provides an interface to native code, such that the streaming computation is expressed in StreamIt, and the remaining code is implemented in languages such as C, C++, or Java. The interface allows native code to invoke the StreamIt pipeline, and vice versa. In addition, we are exploring the use of emerging languages such as X10 [2] as frameworks for providing first class support for domain specific language primitives. We have designed and built a prototype environment that allows StreamIt programs to run as part of the X10 virtual machine. The bridge implements the StreamIt language abstractions using new constructs founded upon the X10 language primitives. We are currently evaluating the productivity merits of this approach, along with the performance potential it affords.

The StreamIt development environment, optimizing compiler, and set of streaming benchmarks are available for download from the project web page at http://cag.csail.mit.edu/streamit.

2. StreamIt Programming Language

In StreamIt, the basic programmable unit is a *filter*. Each filter has an independent address space. Thus, all communication with other filters is via the input and output channels, and occasionally via control messages (see Section 2.2). Each filter contains a *work* function that repeatedly reads data from the input tape, carries out some computation, and writes data to the output tape. A filter may also inspect (*peek*) data on the input without removing them. The peek, *pop* (read) and *push* (write) rates are declared as part of the work function (see Figure 2), thereby enabling the compiler to apply various optimizations and construct efficient execution schedules.

In StreamIt, the application developer focuses on the hierarchical assembly of the stream graph and its communication topology, rather than on the explicit management of the data buffers between filters. StreamIt provides three hierarchical structures for composing filters into larger stream graphs (see Figure 1).



(a) pipeline (b) splitjoin (c) feedback loop

Figure 1. Hierarchical streams in StreamIt.

2.1. Hierarchical Streams

A *pipeline* is a single input to single output parameterized stream. It composes streams in sequence, with the output of one connected to the input of the next. An example of a pipeline appears in Figure 2. The *splitjoin* construct distributes data to a set of parallel streams, which are then joined together in a roundrobin fashion. StreamIt also provides a *feedback loop* construct for introducing cycles in the graph.

The `add` keyword in StreamIt constructs the specified stream using the input arguments. The `add` statement may only appear in non-filter streams. Filters are the leaves in the hierarchical construction, and composite nodes define the encapsulating containers. Thus StreamIt allows for modular design and development of large applications, thereby promoting collaboration, increasing code reuse, and simplifying debugging.

In a splitjoin, the *splitter* performs the data scattering, and the *joiner* performs the gathering. A splitter is a specialized filter with a single input and multiple output channels. On every execution step, it can distribute its output to any one of its children in either a *duplicate* or a *roundrobin* manner. A duplicate splitter (indicated by `split duplicate`) replicates incoming data to each stream connected to the splitter. A roundrobin splitter (indicated by `split roundrobin(`$w_1, \ldots, w_n$`)`) distributes the first w_1 items to the first child, the next w_2 items to the second child, and so on. The splitter counterpart is the joiner. It gathers data from its predecessors in a roundrobin manner to produce a single output stream.

```
float->float pipeline IDCT_2D(int N) {
  // N 1D-IDCT in parallel in the X direction
  add splitjoin {
    split roundrobin(N);
    for (int i = 0; i < N; i++)
      add IDCT_1D(N);
    join roundrobin(N);
  }
  // N 1D-IDCTs in parallel in the Y direction
  add splitjoin {
    split roundrobin(1);
    for (int i = 0; i < N; i++)
      add IDCT_1D(N);
    join roundrobin(1);
  }
}

float->float filter IDCT_1D(int N) {
  float[N][N] coeff = { ... };

  work peek N pop N push N {
    for (int x = 0; x < N; x++) {
      float product = 0;
      for (int u = 0; u < N; u++)
        product += coeff[x][u] * peek(u);
      push(product);
    }
    for (int x = 0; x < N; x++) pop();
  }
}
```

Figure 2. Example StreamIt code for 2D inverse DCT using two 1D transforms.

```
// global variable
float coeff[64] = { ... };

void IDCT_2D(float* block) {
  int i, j, u;
  float product;
  float tmp[64];

  // 1D DCT in X direction
  for (i = 0; i < 8; i++)
    for (j = 0; j < 8; j++) {
      product = 0;

      for (u = 0; u < 8; u++)
        product += coeff[u][j] * block[8*i + u];

      tmp[8*i + j] = product;
    }

  // 1D DCT in Y direction
  for (j = 0; j < 8; j++)
    for (i = 0; i < 8; i++) {
      product = 0;

      for (u = 0; u < 8; u++)
        product += coeff[u][i] * tmp[8*u + j];

      block[8*i + j] = product;
    }
}
```

Figure 3. Example C code implementation of 2D inverse DCT using two 1D transforms.

The stream constructs provide a convenient and natural way to expose pipeline parallelism, as well as data parallel streams. The language also naturally exposes data distribution and communication between streams. An example is shown in Figure 2. It illustrates a parallel implementation of a 2D inverse discrete cosine transform (DCT) using 1D inverse DCTs. This implementation is both data parallel (within the rows and columns) and pipeline parallel (between the rows and columns). A straightforward C implementation of a computationally equivalent inverse DCT is shown in Figure 3. Note that the code structure is similar to the StreamIt version, although it does not explicitly expose the parallelism (i.e., parallelization requires loop analysis and array distribution). The C code also requires explicit array index management, such as the expressions `block[8*i+u]` and `tmp[8*i+j]` which are notably absent in the StreamIt code. The splitter and joiner in StreamIt free the programmer from tedious indexing operations, and enable the compiler to understand and optimize buffer management [8]. The StreamIt implementation is also parameterized such that it is trivial to adjust the size of the inverse DCT. The equivalent parameterization of the C code requires adjustments to the loop bounds and array access expressions.

2.2. Teleport Messaging

A difficult aspect of stream programming, from both a performance and programmability standpoint, is reconciling regular streaming dataflow with irregular control messages. While the high-bandwidth flow of data is very predictable, realistic applications also include unpredictable, low-bandwidth control messages for adjusting computation parameters (e.g., coefficients used for quantization during MPEG encoding and decoding). In StreamIt, such communication is conveniently accomplished with teleport messaging [11]. The idea behind teleport messaging is for a message sender to communicate information via asynchronous method calls to the intended receivers.

The method invocations in the targets are timed relative to the flow of data in the stream. As a result, teleport messaging avoids muddling data streams with control-relevant information. In addition, teleport messaging separates the concerns of the programmer from the system implementation, thereby allowing the compiler to deliver the message in the most efficient way for a given architecture: by exposing the exact data dependences to the compiler, filter executions can be reordered so long as they respect the message timing. Such reordering is generally impossible if control information is passed via global variables.

3. Language Interoperability

In StreamIt, a filter work function may execute whenever the amount of data on the input tape is equal to or greater than the declared input rate (i.e., max of peek and pop rates). The compiler can take advantage of the declared I/O rates to statically schedule filter executions. There are often many legal execution orders for a given stream graph. The compiler can manage the schedule such that it accounts for the buffer requirements between filters and the instruction footprint of the stream graph, and can optimize the output latency or computational throughput [6, 8].

The StreamIt compiler can generate single threaded or multi-threaded code for commodity processors, including computing clusters. We have also developed a Java emulation library to promote research and experimentation in language design and compiler innovation. StreamIt programs can run as standalone applications, but may also integrate with native codes. For example it is possible to use highly optimized libraries for specific computations (e.g., FFTW [4] for convolutions in signal processing). StreamIt provides an interface to native codes which may be viewed as black box components by the StreamIt compiler. Similarly, a native to StreamIt interface allows applications to view StreamIt code as a black box component. Application developers can thus leverage the benefits of StreamIt for computation that is well suited for the streaming paradigm.

Recently, we began to explore how constructs in X10 [2] may be used to express streaming abstractions. X10 is a new high performance programming language for parallel architectures. It features many novelties, of which three are especially relevant to StreamIt. The first is the concept of a *place*. Informally, a place is partition in a global address space, and can encapsulate filter code. The second is the ability to specify the *distribution* of arrays, which is useful to expose the communication between streams, as in the splitjoin containers. The last is the ability to initiate asynchronous method calls via the *async* construct, which provides a facility to implement teleport messaging.

We have designed and implemented a runtime interface to run StreamIt programs as part of an experimental X10 virtual machine. When a program is launched, a StreamIt library constructs the stream graph and performs all filter initialization. The library then iterates through the stream graph to create equivalent X10 stream objects that wrap around filters and tapes. Currently the virtual machine ignores the StreamIt compiler scheduling decisions. Instead, it uses a push model of execution where a filter is run as soon as a sufficient number of input data items appear on its input stream.

4. Concluding Remarks

As computer architectures change from the traditional monolithic processors, to scalable wire-exposed and multicore processors, there is a greater need for portable applications that expose parallelism and communication to enable efficient and high performance executions—while also boosting programmer productivity. StreamIt is a programming language and a compilation infrastructure specifically engineered to naturally expose and leverage stream abstractions that are embodied in modern streaming applications. We have used StreamIt to implement DSP codes (e.g., software radio, beamforming), image and video codecs (e.g., MPEG-2 encoding and decoding), encryption algorithms (e.g., DES and Serpent), and many other applications. The language, compiler, and applications are available for download from the project web page at http://cag.csail.mit.edu/streamit.

The goal of the StreamIt project is to boost productivity for streaming application developers such that they focus on algorithmic innovation rather than on performance tuning. The ability to leverage domain specific language constructs affords optimization opportunities that can deliver high performance from high levels of abstraction.

We believe emerging languages such as X10 can provide a framework to implement domain specific abstractions in a general purpose programming model. We have designed and implemented a prototype bridge to run StreamIt code as part of the X10 virtual machine. As a result, application developers can use the streaming abstractions for the computation that fit that model of computation, while different abstractions can be used to describe other aspects of the computation. A part of our ongoing work is concerned with evaluating the productivity and performance merit of native interfaces to and from StreamIt codes.

Acknowledgements

The StreamIt project is supported by DARPA grants PCA-F29601-03-2-0065 and HPCA/PERCS-W0133890, and NSF awards CNS-0305453 and EIA-0071841. Special thanks to Vijay Saraswat for his help in navigating X10 and answering various questions about the infrastructure.

References

[1] S. Agrawal, W. Thies, and S. Amarasinghe. Optimizing stream programs using linear state space analysis. In *CASES*, 2005.

[2] P. Charles, C. Donawa, K. Ebcioglu, C. Grothoff, A. Kielstra, C. von Praun, V. Saraswat, and V. Sarkar. X10: An Object-oriented Approach to Non-Uniform Cluster Computing. In *OOPSLA Onwards! Track*, 2005.

[3] M. Drake, H. Hoffman, R. Rabbah, and S. Amarasinghe. Mpeg-2 decoding in a stream programming language. In *IPDPS*, 2006.

[4] M. Frigo and S. Johnson. Fftw: An adaptive software architecture for the fft, 1998.

[5] M. Gordon, W. Thies, M. Karczmarek, J. Lin, A. S. Meli, C. Leger, A. A. Lamb, J. Wong, H. Hoffman, D. Z. Maze, and S. Amarasinghe. A Stream Compiler for Communication-Exposed Architectures. In *ASPLOS*, 2002.

[6] M. Karczmarek, W. Thies, and S. Amarasinghe. Phased scheduling of stream programs. In *LCTES*, 2003.

[7] A. A. Lamb, W. Thies, and S. Amarasinghe. Linear Analysis and Optimization of Stream Programs. In *PLDI*, 2003.

[8] J. Sermulins, W. Thies, R. Rabbah, and S. Amarasinghe. Cache Aware Optimization of Stream Programs. In *LCTES*, 2005.

[9] M. B. Taylor, W. Lee, J. Miller, D. Wentzlaff, I. Bratt, B. Greenwald, H. Hoffmann, P. Johnson, J. Kim, J. Psota, A. Saraf, N. Shnidman, V. Strumpen, M. Frank, S. Amarasinghe, and A. Agarwal. Evaluation of the Raw Microprocessor: An Exposed-Wire-Delay Architecture for ILP and Streams. In *ISCA*, 2004.

[10] W. Thies, M. Karczmarek, and S. Amarasinghe. StreamIt: A Language for Streaming Applications. In *Int. Conf. on Compiler Construction*, 2002.

[11] W. Thies, M. Karczmarek, J. Sermulins, R. Rabbah, and S. Amarasinghe. Teleport messaging for distributed stream programs. In *PPoPP*, 2005.