Streams: Emerging from a Shared Memory Model

Benedict R. Gaster

ClearSpeed Technology Plc
S/W Architecture Group
brg@clearspeed.com

Abstract. To date OpenMP has been considered the work horse for data parallelism and more recently task level parallelism. The model has been one of shared memory working in parallel on arrays of a uniform nature, but many applications do not meet these often restrictive access patterns. With the development of accelerators on the one hand and moving beyond the node to the cluster on the other, OpenMP's shared memory approach does not easily capture the complex memory hierarchies found in these heterogeneous systems.

Streams provide a natural approach to coupling data with its corresponding access patterns. Data within a stream can be easily and efficiently distributed across complex memory hierarchies, while retaining a shared memory point of view for the application programmer.

In this paper we present a modest extension to OpenMP to support data partitioning and streaming. Rather than add numerous new directives our approach is to utilize exiting streaming technology and extend OpenMP simply to control streams in the context of threading. The integration of streams allows the programmer to easily connect distinct compute components together in an efficient manner, supporting both, the conventional shared memory model of OpenMP and also the transparent integration of local non-shared memory.

1 Introduction

OpenMP's shared memory model is one of its strongest points, providing a simple view of memory for the programmer. However, to increase memory bandwidth and reduce memory contention many of today's processors have complex memory hierarchies that do not directly fit this model. For example, both IBM's Cell [1] and ClearSpeed's CSX [2] processors have small single cycle memories attached to local processing elements. These memories are not memory mapped into the larger outer memory system and thus are not shared in the conventional sense, rather data is moved to and from shared memory via DMA transfers. These memories are a problem for the OpenMP programmer as there is no easy why to describe the connection between objects in shared memory and corresponding objects in local memory. Moreover, these memories are often small in size, at most in the region of hundreds of kilobytes, and it is often impossible to keep the complete data set in memory at any given time. Data needs to be "streamed".

In this paper we describe a primitive streaming API that when embedded in a modest extension of OpenMP provides for a powerful alternative to the conventional array based parallel programming model. In particular, it is possible to express complicated non-uniform access patterns for streams that are not easily expressed in OpenMP as is. Streams [3,4,5,6] are best described as a declarative interface to conventional C/C++ style data arrays, that provide for a parallel evaluation semantics, standard and user defined scatter/gather access, and a small set of combinators for writing stream computations. Streams are defined and referenced from anywhere within an OpenMP program, with data pushed and pulled across thread boundaries as specified by the user.

We take the rather unconventional approach of assuming a new basic type, i.e. streams, in the base language. However, it should be noted that we are not proposing extending the base language itself, rather we assume that a streaming API is provided as a library in the particular language of choice. There are many examples of streaming APIs and their implementations are well understood and given this it does not seem unreasonable to build upon these developments [3,4,5,6]. The advantage is that given a non-parallel program, written using streams, it is a natural process to add (extended) OpenMP directives to parallelize for a multi-core environment. This scheme is analogous to that of adding OpenMP directives today in the context of arrays.

In this paper we make the following contributions:

– We describe a modest extension to OpenMP's programming model based on the notion of streams. This model provides an alternative to the conventional array approach, conceptually extending OpenMP's memory model to work in the context of non-shared memory.
– We are implementing a prototype of OpenMP extended with streams for an IA-32 and CSX accelerator based system and we outline its current status.
– We report our experience of using the proposed tasking model for OpenMP with streams, highlighting its natural use in this context.

The remaining sections of this paper are as follows: Section 2 discusses related work; Section 3 introduces, by way of example, how streams can be utilized in OpenMP; Section 4 details the streaming API and the extensions to the OpenMP directives; Section 5 outlines our implementation with some early performance results; and finally Section 6 concludes with a discussion on possible directions for the future.

2 Related Work

To our knowledge there has been very little previously published work on extending OpenMP with a notion of streams. One exception to this is the ACOTES project defining a programming model for streams with a corresponding abstract streaming machine [7]. Carpenter et al. propose a new streaming environment consisting of a Stream Programming Model (SPM), implemented as an annotated version of the C programming language, and an Abstract Streaming Machine (ASM), implemented as a cost-model simulator. Their approach is similar to what is proposed in this paper, although we present extensions to both the data parallel and tasking features of OpenMP while they consider only the tasking aspects. To date they have not implemented their approach and work in this area is necessary to better understand how useful it will be in practice.

There exist a number of proposals for mapping the shared memory model of OpenMP to a distributed setting which is closely related [8,9,10]. These approaches have some important advantages over explicit distributed programming models, such as MPI, including they are conservative extensions to OpenMP and retain the shared memory model. However, the drawback is that they retain this model at the expense of restricting control of data partitioning and movement to the system, thus constraining expressiveness. Providing streams as a first class data abstraction retains OpenMP's shared memory model while exposing control of both data partitioning and movement within a distributed memory setting.

Eichenberger et al. use a software cache to abstract the Cell's SPEs local memory, for both code and data, providing a transparent, shared memory, view of memory [11]. The advantage of this approach is no new datatypes need to be introduced and thus no unnecessary source code changes. What is less clear is how well this approach works in the light of more complex memory layouts which may include many levels of indirection. The overhead of maintaining a software cache in this context could easily dwarf any benefits of such an approach.

The streaming library given in Section 4 is a variant of the streams of Open Accelerator [3]. Open Accelerator is a programming environment that supports accelerator specific code with the integration of streaming, allowing the programmer to easily and efficiently connect distinct components of a system. Open Accelerator is itself orthogonal to OpenMP but Gaster et al. show by superimposing Open Accelerator the resulting system provides a powerful SPMD data parallel and tasking programming abstraction for accelerated systems. A key difference between Open Accelerator and this paper is that streams become a data abstraction that OpenMP builds parallelism upon and are integrated into the language, requiring no additional features or support.

Finally, our work on streams has continually built on the ideas of Brook [6] and StreamIt [5], which provide the stream processor abstraction. While these languages do not fit directly with an imperative programming model it is clear that they provide a wealth of resources for the development of streaming techniques in such a context.

3 Overview

When considering adding a new feature to OpenMP, the place to start, at least one would believe, is the current set of existing directives and possibly some new ones. This was our initial approach when looking at streams for OpenMP and we developed a set of additional directives for working with streams along with some new runtime functions and extensions to existing directives. As an example of this approach consider code to perform a sum of squares for an input array, a, given in Figure 1.

```
#define CHUNK_SIZE 5

double sumsq(double a[], int size) {
  double msum = 0.; int i, n;

#if _OPENMP
  omp_stream_set_chunk_size(CHUNK_SIZE);
#endif

#pragma omp stream create(s, a, size, sizeof(float),
                                   LINEAR_FORWARD)

#pragma omp for reduction(+:msum) connect(s:size)
  for (i = 0; i < size; i++) {
    double elem;
#pragma omp stream read(s, elem)
    elem = a[i];

    msum += elem * elem;
  }

  return msum;
}
```

Fig. 1. Streams API directive extensions: sum of squares

Here a stream s is created from array, a, with the directive *stream create*, introducing a new stream object into the OpenMP environment. The stream is later consumed by the conventional parallel *for* construct with an additional clause, *connect*(s : *size*), telling the system to produce a one-to-many stream channel from the controlling thread to its corresponding children, that will perform the work of the loop body. The loop body itself is implemented as though the function was operating over the original input array and makes no direct reference to s. Instead, the *stream read* directive is used to connect the next element of s to the private variable *elem* and the following statement is ignored, i.e. elem = a[i]. While straightforward to modify the compiler to ignore the connect statement, this approach reaches beyond the original intention of OpenMP's directives.

From this description of streams one might reasonably ask what is the benefit of this approach over straightforward arrays? Well one important difference is the use of the runtime function to set a stream's "chunk" size and a related reference to *size* in the *connect* clause. A streams *chunk* size corresponds to the number of elements that will be copied to a particular thread when reading from a stream's channel. The actual pattern for reading *chunk* number of elements is captured in the final argument to the *stream create* directive; in the above example a simple linear-forward pattern is assumed. A consequence of chunks is that each thread must maintain a private buffer of *chunk* size that when empty

```
double sumsq(double a[], int size) {
  double msum = 0; int i;

  Stream * s = Stream.create(a,size,CHUNK_SIZE,LINEAR_FORWARD);

#pragma omp parallel reduction(+:msum) connect(s)
  {
    while (!s->endOfStream(s)) {
      double x = s->getNextElement();
      if (s->streamActive()) {
        msum += x * x;
      }
    }
  }

  s->destroy();

  return msum;
}
```

Fig. 2. Streams API part of base language: sum of squares

is refilled with *chunk* or less elements by making a read request to the source stream. The observant reader may now be asking but if *chunk* $\neq 1$ then the loop will be parallelized across *size* threads, but only the first *size*/*chunk* threads will actually read elements of the stream and even worse the active threads will process only the initial chunk element. Fortunately, this is easily over come by requiring that the user link the controlling bounds variable, *size* in this case, when connecting a stream, which is then used to control the number of iterations.

It should be clear that a compiler is free to implement OpenMP with or without the streaming extensions, assuming it at least parses the *connect* clause, while preserving the semantics of the program. This is, of course, in keeping with the original design ethos of OpenMP, allowing both incremental parallelization and programs execute correctly, albeit often with a slower execution time, if the program is not parallelized at all. The problem with such an approach is that OpenMP was not designed on top of a language with native stream support and adding them explicitly to OpenMP is not only clumsy but it feels like using a bulldozer to crush a nut. For C++ and Fortran arrays are the basic type on top of which OpenMP builds parallelism but with the introduction of task level parallelism this no longer needs to be the case and it should be possible to introduce other parallel data types, including streams. This then leads us to the question: what if streams were provided as a basic datatype?

If streams are provided as an abstract type with a corresponding API, then application programmers could write stream programs with or without OpenMP directives. This meets our goal that any OpenMP program can be compiled and executed correctly; even in the presence of a compiler that does not support OpenMP. Such an approach does not preclude the need to add streams to OpenMP, rather

```
    while (!s->endOfStream(s))
#pragma omp task capturevalue(s)
    {
      double x = s->getNextElement();
      if (s->streamActive()) {
        msum += x * x;
      }
    }
```

Fig. 3. Combining streams and tasks

it provides the foundations for a more modest extension. Figure 2 shows how the sum of squares example might be written with this approach.

Intuitively, a stream s is created from an input array a and the parallel regions creates a team of threads, as is normal, with the only difference that each thread t_i having a corresponding private stream s_i that is "connected" to the input stream s. The behavior is similar to that of a variable marked as private, except instead of copying s locally within the thread's stack, a connection is made between the private s_i and the stream s of the shared enclosing scope. On entering the loop, for a particular thread, the private stream requests a new chunk and if data greater than zero and less than or equal to the number of elements in a chunk is received then the request returns the first of the elements for the call *getNextElement*. If no elements at all are received, then the call to *streamActive* returns false and the sum is not executed and finally the loop will terminate, otherwise the element is squared and added into the running total and the process repeats.

At first viewing the call to *streamActive* may seem unnecessary but there are actually two reasons for its inclusion. Firstly, what happens in the time between the call to *endOfStream* and *getNextElement*? In fact anything and in particular as we are running in parallel the stream may get locked and read by some other thread in the farm and thus leave the call undefined. Secondly, if we are to support SIMD or predicated processors, where conditionals do not necessarily imply control flow, then *streamActive* can provide functionality to disable or enable particular processing elements.

Of course, in the case when the amount of work on each element of a chunk is large the while loop itself can be parallelized with task parallelism. For example, the loop of Figure 2 might be expressed as in Figure 3.

4 Extending OpenMP with Streams

In this section we introduce a streaming API and a small addition to OpenMP's parallel fork-join and producer-consumer threading models.

First we define streams as declarative representations of more conventional random access C++/Fortran arrays. Random access to streams is not allowed, and consequently no index operator exists; instead the user can define a gather/

```
template<typename T> class Stream {
public:
  // Creating and destroying streams
  static Stream<T> * create(T * p, int in_stream_size,
                              int chunk_size,
                              tuple<StreamAccess, TargetISA> *t);
  static Stream<T> * create(Stream<T> *);
  void destroy(void);
  // Reading and writing streams
  T getNextElement(void);
  T * getNextChunk(void);
  void writeNextElement(T&);
  // Stream info
  void flush(void);
  bool endOfStream(void);
  bool streamActive(void);
  int numChunks(void);
  int chunkSize(void);
  // Stream reduction
  template<typename U> U reduce(function<U (U, U)>);
private:
  // constructor, destructor, etc...
};
```

Fig. 4. Steaming API (C++ variant)

scatter style access, using either a set of statically defined patterns or dynamically using streams themselves.

The streaming interface is split into four components: types for streams and access patterns; functions for creating and destroying streams; functions to read and write streams; and functions returning stream characteristics. A stream is created with an array forming the data stream, the size of input, and a list of access patterns. It may not at first be obvious why a list of access patterns is required, rather than just a single value. A single access pattern works fine in the case that a particular stream is destined for a single source but consider the case when a particular stream is distributed across a number of different ISAs, e.g. accelerator cores, which may themselves spread the received data across any number of internal cores. It becomes necessary to associate a particular target ISA ($TargetISA$) with a corresponding stream access, allowing the stream implementation to know at which level a distribution is to be applied.

The streaming functions should be reasonably self-explanatory and the complete set is given in Figure 4.[1] A full description of the streaming API is beyond the scope of this paper and the interested reader is pointed to Gaster et al. for a detailed presentation [3].

For OpenMP itself Figures 5(a) and 5(b) show the extensions necessary for the parallel fork-join and producer consumer directives. As discussed in the previous

[1] For simplicity we use C++ as our base language.

#pragma omp parallel [clause[[,]clause]...]
structured-block

where clause can be one of
firstprivate(list)
private(list)
shared(list)
reduction(operator:list)
num_threads(integer-expression)
connect(list)

(a) Fork-join threading model

#pragma omp task [clause[[,]clause] ...]
structured-block

where clause can be one of
captureprivate(list)
private(list)
shared(list)
switch
connect(list)

(b) Producer-consumer threading model

Fig. 5. Streaming extensions to OpenMP pragmas

section this is reduced to a single additional clause connect, capturing the notion that a stream defined at an outer scope is to be joined (one-to-many) to the parallel region's gang of threads. The stream creation method

```
static Stream<T> * create(Stream<T> *);
```

is provided for this and *joins* a stream to the calling region, which in this case will be a thread in the parallel region. The resulting stream is then used in place of the referenced stream within the structured block. On exit from the parallel region the created stream(s) must be destroyed. For output streams this will cause the streams to be flushed and data will be moved to the corresponding stream of the outer scope.

An important consideration when describing a new API for OpenMP must be how easily it can be expressed in C, C++, and FORTRAN. We choose to specify the streaming library in C++ as the parametric polymorphism provided by templates leads to a simple definition. However, while maybe not as compelling when expressed in C or FORTRAN it is straightforward. With careful use of the preprocessor it is possible to generate much of the boiler-plate code necessary for parameterized stream types while retaining most of the generic approach offered by C++. For example, the code in Figure 6 implements a C macro, $STREAMING\_TYPE(typ)$, that when instantiated generates the set of steaming functions for streams of element type *typ*. While at times not the most elegant of approaches, it does mean that it is possible to retain type safety and makes it applicable in the context of FORTRAN.

5 Evaluation

The streaming environment described in this paper has already been implemented within ClearSpeed for developing applications for an IA-32 system with any number of CSX accelerators. The implementation is factored into two parts: a source-to-source compiler, based on the Barcelona Supercomputing Center's Mercurium compiler [12]; and runtime components for both the IA-32 and CSX.

```
#define STREAMING_TYPE(typ)                                    \
  static typ * get_next_ ## typ ## _chunk(stream s) {          \
    return ((typ *) get_next_chunk(s));                        \
  }                                                            \
  static typ get_next_ ## typ ## _element(stream s) {          \
    return (* ((typ *) (host_get_next_element_p(s))));         \
  }                                                            \
  static void write_next_ ## typ ##                            \
                          _element(stream s,typ x) {           \
    write_next_element_p(s,(char *) &(x));                     \
  }                                                            \
  static void                                                  \
  init_ ## typ ## _stream (stream s,                           \
           typ * p,                                            \
           typ * buf0,                                         \
           typ * buf1,                                         \
           int in_stream_size,                                 \
           int out_buf_size,                                   \
           stream_type t) {                                    \
    init_stream(s, (char *) p, (char *) buf0, (char *) buf1,\
                in_stream_size,out_buf_size,sizeof(typ),t); \
  }

STREAMING_TYPE(int);
STREAMING_TYPE(float);
STREAMING_TYPE(double);
```

Fig. 6. Using C's preprocessor to generate a stream API

The streaming API itself is a standard ClearSpeed product for both IA-32 and CSX and required no modifications. The OpenMP compiler expects a single source input, expressed in C++ with OpenMP SIMD accelerator regions [13], that is processed to produce corresponding IA-32 (C++) and CSX (C^n [14]) code, compiled by respective compilers.

For this paper we have evaluated our implementation against a small number of representative benchmarks for performance evaluation. Rather than considering the performance ratio between a native IA-32 implementation and a CSX accelerated system—it is easy to show performance improvements for applications with CSX assistance [2,13]—we considered differences for implementations in OpenMP with and without streams.

The performance figures for each of the selected benchmarks are given in Figure 7[2]. The results themselves are of an early nature but they are very encouraging as is evident, in particular, from the results for the FFT (performing 10,0000 1k and 2k 2D FFTs), sum (sum of squares of one million doubles), and

[2] All benchmarks were compiled with GCC 4.1, optimization level -03, and ClearSpeed's latest CSX SDK (3.0).

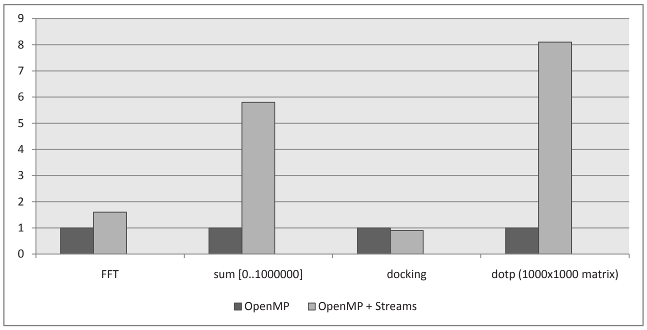

Fig. 7. Performance for OpenMP + Streams vs OpenMP codes

dotp (dot product of 1k vector) which all show a performance improvement over the OpenMP versions, the latter showing a 5x and 7x speedup.

6 Conclusion

In this paper we have presented an approach to extending OpenMP's shared memory model for systems where not all memory is shared. Taking the (possibly) surprising step of assuming a primitive notion for streams it is possible to extend OpenMP in a modest fashion. The programmer is provided with a powerful parallel programming abstraction with the ability to describe irregular data access across distributed memory hierarchies.

In practice the streaming extensions are small and maintain the semantics of existing programs while providing a natural approach to data partitioning not present in OpenMP as it stands. Streams themselves provide a natural parallel programming abstraction and it seems only sensible to consider their application in a parallel programming language such as OpenMP.

With the introduction of streams as alternative containers to arrays, task level parallelism may (often) be more convenient than the conventional data parallel constructs. This is probably due, in part at least, to the fact that OpenMP's original parallel constructs were designed with large scale data-parallelism in mind. With the introduction of task level parallelism it is possible to consider irregular data access that fits well with the produce-consumer style model that arises naturally when working with streams.

6.1 Future Work

Multimedia codecs, such a MEG2 and software radio, show a high amount of data-parallelism and initially seem like a good fit for the data parallel constructs of OpenMP. The problem is that often fine-grained control flow and data communication is required that makes simply loop parallelization difficult and task parallelism finds a better fit. These applications also fit well with gather/scatter and compute style semantics and an interesting area of future work is to develop implementations for a selection of multimedia codecs using the model described in this paper.

As the streams presented in this paper are treated as declarative objects and their computations can be specified using a small set of combinators. It is possible, in many cases, for the compiler to optimize the generation of possible intermediate streams statically. A more detailed discussion of these and other stream optimizations is outside of the scope of this paper and the interested reader is pointed to work on Open Accelerator for more information [3]. To date we have not evaluated the use of these kinds of optimizations in the presence of OpenMP but believe this to be an interesting avenue for future work.

The shared memory model of OpenMP is known to have problems when scaling to simultaneous multi-threaded (SMT) [15] processors. In particular, when OpenMP applications are executed on SMT architectures many different forms of interference between threads has been reported [16]. While not the motivation for the work described in this paper we believe that streams may provide an approach to parallel data access that avoids many of the data interference issues on SMT systems and this is an important area of future work.

Finally, it possible that an implementation of OpenMP built on top of a streaming API could implicitly connect streams to parallel regions without the need for the *connect* clause at all. One problem with this approach comes when considering extending the *connect* clause to capture information on intermediate stream production, which in practice could be optimized away, see Gaster et al. [3] for an example. In this case an explicit stream connection provides vital static information that may otherwise be hidden from the compiler.

References

1. Pham, D., Asano, S., Bolliger, M., Day, M.N., Hofstee, H.P., Johns, C., Kahle, J., Kameyama, A., Keaty, J., Masubuchi, Y., Riley, M., Shippy, D., Stasiak, D., Suzuoki, M., Wang, M., Warnock, J., Weitzel, S., Wendel, D., Yamazaki, T., Yazawa, K.: The design and implementation of a first-generation CELL processor. In: Solid-State Circuits Conference, 2005. Digest of Technical Papers. ISSCC, vol. 1, pp. 184–592. IEEE International, Los Alamitos (2005)
2. ClearSpeed Technology Plc: White paper: CSX Processor Architecture (2004)
3. Gaster, B.R., Lacey, D., Sumner, B.: Open Accelerator: programming at the edges (submitted 2007)
4. Tarditi, D., Puri, S., Oglesby, J.: Accelerator: using data parallelism to program GPUs for general-purpose uses. In: ASPLOS-XII: Proceedings of the 12th in- ternational conference on Architectural support for programming languages and operating systems, pp. 325–335. ACM Press, New York (2006)
5. Thies, W., Karczmarek, M., Amarasinghe, S.P.: StreamIt a language for streaming applications. Computational Complexity, 179–196 (2002)
6. Buck, I.: Brook: A Streaming Programming Language. Stanford University (2001)

7. Carpenter, P., Rdenas, D., Martorell, X., Ramrez, A., Ayguad, E.: A Streaming Machine Description and Programming Model. In: Vassiliadis, S., Bereković, M., Hämäläinen, T.D. (eds.) SAMOS 2007. LNCS, vol. 4599, pp. 107–116. Springer, Heidelberg (2007)
8. Huang, L., Chapman, B., Liu, Z.: Towards a more efficient implementation of OpenMP for clusters via translation to Global Arrays. Parallel Comput. 31(10-12), 1114–1139 (2005)
9. Basumallik, A., Min, S.J., Eigenmann, R.: Programming distributed memory systems using openmp. In: 12th international workshop on High-Level Parallel Programming Models and Supportive Environments (2007)
10. Sato, M., Harada, H., Hasegawa, A., Ishikawa, Y.: Cluster-enabled OpenMP: An OpenMP compiler for the scash software distributed shared memory system. Sci. Program 9(2,3), 123–130 (2001)
11. Eichenberger, A.E., O'Brien, J.K., O'Brien, K.M., Wu, P., Chen, T., Oden, P.H., Prener, D.A., Shepherd, J.C., So, B., Sura, Z., Wang, A., Zhang, T., Zhao, P., Gschwind, M.K., Archambault, R., Gao, Y., Koo, R.: Using advanced compiler technology to exploit the performance of the Cell Broadband EngineTM architecture. IBM Syst. J. 45(1), 59–84 (2006)
12. Barcelona Supercomputing Center: The NANOS Environment
13. Bradley, C., Gaster, B.R.: Exploiting loop-level parallelism for SIMD arrays using OpenMP. In: International Workshop on OpenMP (2007)
14. Lokhmotov, A., Gaster, B.R., Mycroft, A., Stuttard, D., Hickey, N.: Revisiting SIMD programming. In: 20th InternationalWorkshop on Languages and Compilers for Parallel Computing (2007)
15. Curtis-Maury, M., Ding, X., Antonopoulos, C.D., Nikolopoulos, D.S.: An evaluation of OpenMP on current and emerging multithreaded/multicore processors. In: First International Workshop on OpenMP (2005)
16. Zhang, Y., Burcea, M., Cheng, V., Ho, R., Voss, M.: An adaptive OpenMP loop scheduler for Hyperthreaded SMPs. In: International Conference on Parallel and Distributed Computing Systems, pp. 256–263 (2004)