Compiler Manipulation of Stream Descriptors for Data Access Optimization

Abelardo López-Lagunas
ITESM Campus Toluca
abelardo.lopez@itesm.mx

Sek M. Chai
Motorola Labs
sek.chai@motorola.com

Abstract

*Efficient data movement is one of the key attributes for high performance computing. This paper advocates the use of **stream descriptors** to convey memory access patterns from the programmer to the compiler. This explicit separation of computation and data movement enables the compiler to manipulate the stream descriptors to match the system's interconnect capabilities. Data movement is optimized by manipulating stream descriptors to target specific optimizations such as bandwidth management and buffer allocation. In this paper, bandwidth improvements are shown for an example system performing video analysis using computer vision methods. The system includes key hardware mechanisms that use stream descriptors to prefetch and align data for stream processors.*

1. Introduction

In today's System-on-a-Chip (SoC), multiple specialized computing units are used to delegate heavy computation load from conventional processors. These computing units operate on vast amounts of data, making data movement the performance bottleneck. Scheduling data movement is a challenge for a compiler, and even the best of programmers. This is because traditional programming models lack the ability to orchestrate data movement and computation among the computing units and memories. It is possible to rely on hardware circuits to dynamically predict the access patterns and pre-fetch the required data from main memory into faster local memory. This approach has limited performance because the access patterns may be difficult to predict in the general case. Also, the access patterns may change from one hierarchy level to the next, requiring pre-fetch hardware for each level of the hierarchy. The associated pre-fetch circuits consume energy and chip area that can otherwise be allocated to actual data processing.

This paper advocates the stream programming model for its ability to explicitly express data movement separately from data-parallel computation. This separation of data movement from computation enables the programmer to express data access patterns through a standardized application program interface (API). A compiler can then use the description of data movement to schedule data transfers ahead of the computation without performing complex dataflow analysis.

This paper has two main contributions. The first contribution is a description of memory access patterns, called *stream descriptors*, capable of capturing the shape and movement of the data throughout the interconnect and memory hierarchies. The second contribution is a collection of stream descriptor manipulations that optimize data movement, locally as well as globally, to better match the hardware capabilities. These optimizations can lead to better latency resulting in an efficient usage of the interconnect and memory hierarchies in the SoC.

The structure of the paper is as follows: Section 2 presents the related work; Section 3 introduces the stream descriptor notation, gives some examples on stream descriptor usage, and presents some hardware mechanisms and their interface to the stream descriptors; Section 4 shows examples of stream manipulation methods; Section 5 defines the benchmark kernels and the evaluation setup used to assess the benefits of stream manipulations; Section 6 shows the improvements introduced by the stream manipulations; Section 7 presents the conclusions and directions of future research.

2. Related Work

Stream programs exhibit data parallelism, regular communication patterns, and short data lifetimes [1]. In this model, large sequences of ordered data (streams) are passed through a series of computation kernels (filters). The programmer separately defines the data access and computation in an explicit manner, thereby facilitating the compiler's ability to schedule both in hardware. Examples of stream programming languages include StreamIt [2], Brook [3], and StreamC/KernelC [4]. The proposed stream descriptors extend these programming languages with an interface that

facilitates the description of the data structures' shape and their access patterns for each computation kernel. Because the data streams exhibit regular access patterns, the compiler can optimize data movement by analyzing stream descriptors along with information that models the underlying hardware. The salient feature in stream descriptors is the ability for the compiler to manipulate different fields of the stream descriptors to match different hardware structures, while maintaining the programmers' intent. Stream descriptors can also be used to target new streaming architectures that use the stream model of computation [5, 6, and 7].

There is a significant body of research in data pre-fetching techniques and reordering of memory accesses to improve performance. Readers are referred to [8] for an overview of compile-time and run-time techniques. The proposed stream descriptors use compile-time techniques to optimize the access order of each data element in the interconnect hierarchy. Stream descriptors can be used to describe the shape of data in memory, such as the regular data access patterns occurring in tight loops, thus enabling data prefetching and alignment. The compiler can use information about the physical attributes of the system when manipulating stream descriptors to optimize desired performance metrics accordingly. The descriptors can also be modified at run-time if the underlying hardware provides run time support.

There are numerous compiler techniques that extract data parallelism by removing data dependencies from loop structures [9, 10]. The compiler applies a series of code transformations to eliminate dependencies, and then generates sequences of load/store instructions based on new access patterns of the transformed loop. This means that data transfer depends on the access pattern inferred by the compiler from the loop structure. In contrast, stream descriptors enable complex access patterns that are not easily discernible from nested loop structures. Stream descriptors also decouple memory address generation and alignment from computation such that the processor and memory hierarchy can be optimized separately.

There are different ways to describe data shapes. This paper uses an extension on the format presented in [11]. Stream descriptors have been used to optimize transfers from I/O devices [12] and from memory [13] without any manipulation. Similar techniques that describe the shape of memory access have also been used for trace generation [14, 15]. This paper explores further optimizations by manipulating the stream descriptors to better match the underlying memory and interconnect hierarchies.

3. Stream Descriptors

From the point of view of the stream processor, data appear as a stream of contiguous elements even though they may be scattered throughout memory. Stream data elements are grouped into logical units, or records, that can be processed in parallel. Unlike vector data, stream records can represent more complex structures, patterns and shapes [16]. In the following subsections the proposed stream descriptor API is presented and some examples are given.

3.1 Stream Descriptor Definition

Stream descriptors have a programming interface that allows the programmer to describe the shape of data structures and their movement in the memory hierarchy. In this paper, stream descriptors are represented by the 7-tuple *(Start Address, Offset(), Stride, Span, Skip, Type, Element_Count)* where:

- **Start Address** is the memory address of the first element of the data stream.
- **Offset()** represents a user-defined function that computes the displacement of the *next* stream record from the base address.
- **Stride** is the spacing between two consecutive stream record elements. The units for the spacing are in *data elements*.
- **Span** indicates how many elements are gathered into the stream record before the *skip* displacement is applied.
- **Skip** is the displacement in data elements that is applied between groups of *Span* elements.
- **Type** indicates how many bytes are in each data element, for example 8-bit data is associated with a type value of one, 16-bit of data associates a value of two and so on.
- **Element_Count** indicates how many stream elements are in the data stream.

The *Stride*, *Span*, *Skip*, and *Type* fields define the shape of the data structure or stream record. These fields are manipulated by the compiler to match the functional units in the hardware to the computation kernel requirements and to match the capabilities of the interconnect hierarchy and memory bandwidth in the system.

The *Offset()* and *Element_Count* fields describe how the stream records are assembled into a stream. The access pattern of stream records is expressed as memory address displacements from a base address, which is updated per record by the *Offset()* function. The function can be used to describe either regular or irregular access patterns and can be synthesized as part of the address generation circuit when reconfigurable hardware is used; otherwise, it represents the sequence

of instructions used to program the Direct Memory Access (DMA) unit. In both cases, the compiler must be aware of the target platform to produce the appropriate code or hints for the hardware synthesis process. Section 3.4 presents an example of such a DMA unit.

The *Offset()* and *Element_Count* fields may not be needed for simple stream record movement, such as traversing images one pixel at a time, but keeping them in the API gives the programmer the ability to describe complex movement of data structures.

3.2 Stream Descriptor Examples

Consider a kernel that operates on stream records of three data elements. Suppose that there is a gap of a single space between each data element, and that there is a gap of three spaces between each group of three elements. In this case the *stride* is two, the *span* is three, and the *skip* is four, as shown in Figure 1.

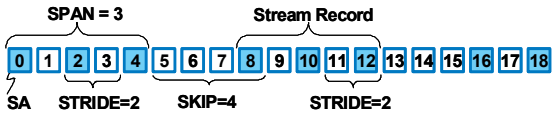

Figure 1. Stream record components

The *stride* and *skip* displacements can be positive or negative, but the *span* is always positive. Additional displacements can be included to describe more complex access patterns, but the template described above allows the description of two-dimensional sub-arrays within larger arrays (useful in imaging applications), uniform sub-samples of arrays, circular access of a sub-array (using negative *skip* values), or a combination of them

It is possible to have more than one representation of the same access pattern, but this is not important, as the compiler will manipulate this representation to better match the capabilities of the memory and interconnect hierarchies.

As an example of the *Offset()* function consider a two-dimensional sub-array which comprises a stream record. The shape of the next stream record has the same dimension, but the *Start_Address* includes an additional displacement defined by the *Offset()* function. The *Offset()* function can express any arbitrary sequence that describe locations of stream records throughout memory.

3.3 Hardware Parameters

To perform stream descriptor manipulation, the compiler needs information about the physical attributes of the memory hierarchy and its associated interconnect. Examples of these physical attributes are bus width, latency, capacitance, operation frequency, burst capability, and internal buffering. A complex system may consist of a hierarchy of busses, DMA units, local memories, and bus bridges. The stream descriptors are manipulated by the compiler to match the capabilities of each of these elements to optimize the flow of data. Data transfers are performed by the DMA units, but it is also possible for some high-bandwidth peripherals to have their own DMA circuits.

3.4 Stream Descriptors and DMA engines

The DMA units use stream descriptor information to perform address generation, byte alignment, data ordering, and interfacing to their respective buses. Depending on the attached peripheral, a DMA unit may have one or more input and output stream modules. An example of a DMA unit with two input stream modules and one output stream module is shown in Figure 2. An arbiter unit with a bus bridge handles simultaneous requests from the input and output modules. A control register unit stores the stream descriptors as presented in Section 3.1.

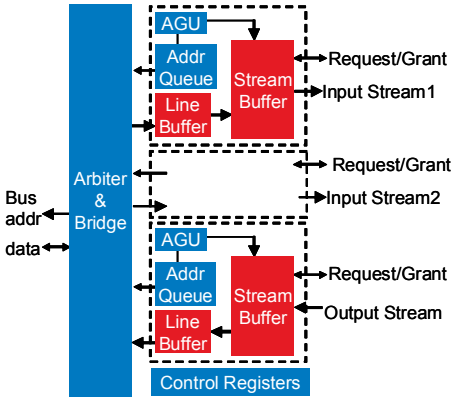

Figure 2. DMA unit block diagram

The input stream module has three key components: Address Generation Unit (AGU), Line buffer, and Stream Buffer. The AGU produces bus addresses based on stream descriptors. Pending requests from the AGU are stored on an address queue. The next bus address is selected from the address queue such that there are no duplicated bus addresses. Once the request has been granted by the bus, the data elements are stored in a line buffer. A stream buffer then selects and aligns stream elements based on the order required by the streaming peripheral.

The output stream module has similar components but data flow in the opposite direction. In this case, data elements are selected from the stream buffer and placed in a line buffer so that the bus interface can proceed with a bus transfer. Bus burst transfers are

used whenever possible to move data in and out of the stream modules.

The configurations of the internal components for the input and output stream modules vary according to the target streaming peripheral. Readers are referred to [17] for more details on the DMA unit.

4. Stream Descriptor Manipulation[1]

This paper presents two stream manipulation operations. The first one changes the stream descriptor fields to match the bus width at each stage in the interconnect hierarchy. The second one merges two stream descriptors into one to better match the interconnect capabilities.

In both stream descriptor manipulations the compiler takes a description of the interconnect hierarchy capabilities and optimization goals to guide the manipulation. The optimization goals can be energy consumption, reduced number of transfers, bus utilization, buffer usage of the bus bridges, etc.

4.1 Stream Descriptor Bus Width Matching

Most systems optimize bandwidth by matching the width of the busses between elements of the memory hierarchy. For example, the main bus matches that of the cache line width. To reduce capacitance, however, the width is reduced for the other system buses, or across the interconnect hierarchy. This implies that the programmer must be aware of these details to perform efficient data transfers.

The first manipulation takes the initial stream descriptor and changes the *stride, skip* and *span* fields to match the bus width across the entire hierarchy. First, the *type* field is matched to the bus width and then the *stride* and *span* fields are divided by the new *type* value. If the division yields a remainder the *stride* and/or *span* fields are rounded up. When the access pattern does not match evenly with the bus width, some additional data elements are fetched but not used by the destination. In this case, the unused data elements are discarded by the DMA engines or by the bus bridges, thus reducing the bandwidth requirements for the hierarchy levels that are closer to the destination. With compiler-driven manipulations of stream descriptors, the programmer does not need to know the bus widths throughout the hierarchy, and the software for the streaming computation becomes architecture independent.

For example, assume that the high-bandwidth bus has a width of 32 bits and the programmer uses one of the hardware accelerator (HA) to process 8-bit data with an access pattern indicated by the stream descriptor (5, *Offset()*, 100, 4, -299, 1, *N*). Also assume that the width of the main bus is 128-bits. The compiler can manipulate this descriptor to match the capabilities of the high-bandwidth bus by indicating to the DMA unit that the data needed by the HA is (5, *Offset()*, 25, 4, -74, 4, *N*). The compiler can also program the bridge and the memory controller so that the stream descriptor from the point of view of the memory controller is (5, *Offset()*, 7, 4, -20, 16, *N*), in both cases the manipulation of the descriptors exploit the bus width and data locality. Note that the *Offset()* function is not affected by this manipulation. Section 6 shows the ability of the DMA units to deliver packed data to the accelerators.

4.2 Stream Descriptor Merging

This operation merges two stream descriptors into one and is useful when two streams have references to common elements in the same regions in memory. For example, assume that two HAs have the access patterns shown in Figure 3. For this case, the compiler generates a stream descriptor that merges the access patterns of both accelerators, which is then used to program the DMA engine in a bus bridge. By combining both descriptors, the bus bridge will generate fewer requests to the memory controller. Note that the compiler generates the load instructions for the two addresses (address one and two, as shown in Figure 3) that do not appear in the new stream descriptor. The DMA engine in the bus bridge splits the incoming stream from the memory controller and delivers two streams of data to the respective HAs.

This manipulation starts by modifying the *type* field in both descriptors so that it is in bytes. The start address of the output stream is the smallest of the start addresses of the input streams. In the simplest case, both stream descriptors have the same values for each field and only differ in the start address. Otherwise, the input streams are analyzed to find out the largest span that results from combining data elements from both input streams. This step generates the *span* field for the output stream and captures the overlap between the input streams. The *stride* field for the output stream is set to one and the *skip* field for the output stream is just the displacement between those elements that were grouped by the *span* field.

[1] A patent is pending that claims aspects of items and methods described in this paper.

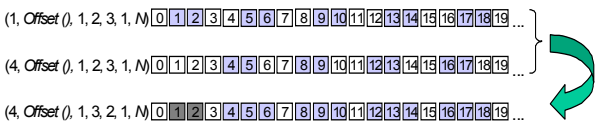

Figure 3. Merging of Stream Descriptors

The efficiency of stream descriptor merging depends on the amount of overlap between the two original streams. In Section 6, we explore the bandwidth savings for different amounts of overlap.

The stream descriptors disassociate memory accesses from computation, enabling the compiler to schedule data transfers ahead of the actual computation. As a result, the stream memory requests do not have to occur at the same time for the compiler to merge stream descriptors. Furthermore, the compiler can perform more than one manipulation to optimize data transfers.

5. Stream Descriptor Evaluation

5.1 Benchmark Kernels

A significant number of image processing and computer vision algorithms are inherently data parallel as they apply the same computation to each pixel in the image. Applications in these domains are built using well-known kernels that perform compute intensive tasks. These kernels manipulate image data in a predefined sequence, making them ideal candidates for the stream computation model. Some of the most used kernels are convolution and morphological filters, image segmentation, region labeling, and pattern matching.

Convolution filters are applied to the entire image as tiles, where each tile is the same size as the convolution mask. The mask is square with an odd number of elements (i.e. 3x3, 5x5, 7x7, etc). Morphological filters operate in a similar way but can have linear or rectangular masks. The pattern-matching kernel is implemented using a template-matching algorithm, which applies a series of templates in sequence to the entire image, in a similar way as the convolution filters do. Image segmentation is used to separate the objects of interest (foreground) from the rest of the scene (background). The benchmark kernel uses a global threshold because it is the most common algorithm. Once the image has been segmented, region labeling assigns a label to each of the objects of interest, or regions, in the image. The benchmark kernel implements the simplest algorithm, which traverses the image twice, first row-wise and then column-wise in reverse order.

The following characteristics are common to the above kernels and have been translated into stream descriptors for evaluation:

- **Dimensionality and size of the stream element**: Stream elements can represent 2D or 1D masks. For a square 2D mask the input stream descriptor is (*SA*, *Offset()*, 1, *m*, (*W* – *m*), 1, *WxH*) with *SA* being the start address of the image, *m* being the width/length of the 2D mask, and *W* and *H* the width and height of the image. The *Offset()* function can describe either row-wise or column-wise traversal. A 1D mask has the input stream descriptor (*SA*, *Offset()*, 1, *l*, (*W* – *l*), 1, *WxH*) where *l* is the length of the 1D mask. Section 6 presents results for filters of dimensions 1x3, 3x1, 1x5, 5x1, 3x3, and 5x5.
- **Traversal order**: The image can be traversed normally starting from the first pixel until the last pixel is reached, or backward starting from the last pixel to the first pixel. The choice in traversal order decides whether the offsets are added or subtracted from the start address.
- **Row-wise or Column-wise traversal:** The image can be traversed in either row-major or column-major form. The *Offset()* function is used to describe the traversal. For row-wise traversal, once all elements of the stream record are gathered, a counter in the *Offset()* function is incremented by one and added to the *Start Address* (SA). For column-wise traversal, two counters in the *Offset()* function can be used, one for rows and one for columns. After the elements from the stream record are fetched, the displacement is incremented by the width of the image and the row counter is increased by one, unless the stream record contains the last element in the column. If this is the case, the column counter is incremented by one, the row counter is reset to zero, and the displacement is set to the column counter. In both traversals *Element_Count* is equal to the total number of pixels in the image.

All the above kernels write one pixel at a time, and thus the output stream descriptor is the same (*SA*, *Offset()*, 1, 1, 1, 1, *WxH*) where the *Offset()* function implements a row-major traversal.

5.2 Evaluation Setup

An integrated simulation platform has been built to show the viability of the stream descriptors by simulating the bus and memory performance during data transfers. For our purposes, the actual computations in these stream applications are not important. Instead, our focus is on the access pattern generated by the benchmarks described in this paper using stream descriptors. By simulating only the access patterns of the benchmark applications we reduced

simulation time significantly, enabling a more detailed exploration on the impact of stream manipulation.

The platform is presented in Figure 4. This platform was built using Verilog HDL to accurately observe system behavior.

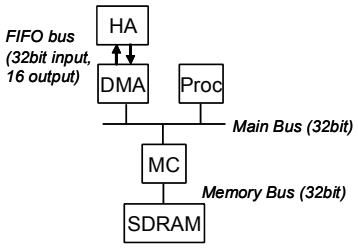

Figure 4. Simulation platform of a system using stream descriptors

In our simulation platform, the DMA unit is designed specifically to use stream descriptors and initiate all memory transfers. The DMA unit, which is described in Section 3.4, has an address generation module that produces memory addresses according to the preloaded stream descriptors, thereby pre-fetching data elements before they are needed. The DMA unit has one line-buffer per stream that is sized to the bus width. It also has FIFO buffers to store stream elements in the order requested by the HA. Unlike regular DMA designs, this DMA unit organizes the data received from memory into streams for the HA. It also writes the stream data produced by the HA by grouping multiple stream elements into a line buffer before initiating write transfers.

The memory controller (MC) and processor model (Proc) are open-source designs [18] that are modified only to include circuits to measure performance. The main bus follows the open-source Wishbone protocol [19] and runs at 200MHz. The SDRAM module is the Micron memory model for a 64Mb SDRAM (2M x 32 x 4banks, 100MHz, CAS latency of 2 cycles) [20]. This memory controller, bus protocol, and memory model do not have specialized functionality and were chosen to provide the readers with a baseline reference of performance.

The configuration of the components in the simulation platform is chosen to represent different system configurations. As an example, the HA and DMA units shown in Figure 4 can represent an independent memory mapped peripheral or accelerator on the main bus. The same units can also represent a tightly coupled coprocessor that has direct access to memory without interaction through the cache. Finally, the HA unit in Figure 4 can also represent memory traffic generated by another bus segment. The abstraction allows for a simplified simulation platform in which simulation times are greatly reduced without sacrificing the investigation in the use and scalability of stream descriptors in the entire interconnect hierarchy.

6. Evaluation Results

We present our results in two categories: a baseline bandwidth of the individual stream descriptors and results from merging two stream descriptors. In Figure 5, we show our results as the effective bandwidth in terms of bytes per transfer. The name of the kernel indicates the dimensionality of the filter mask (1D, 2D), its size (3x3 or 5x5), its access (c for column-wise), and its traversal order (forward or backward). This illustrates the ability of the DMA unit to utilize stream descriptors to deliver packed requests on the bus, and is calculated from the total number of bytes and total number of busy cycles required for data transfer.

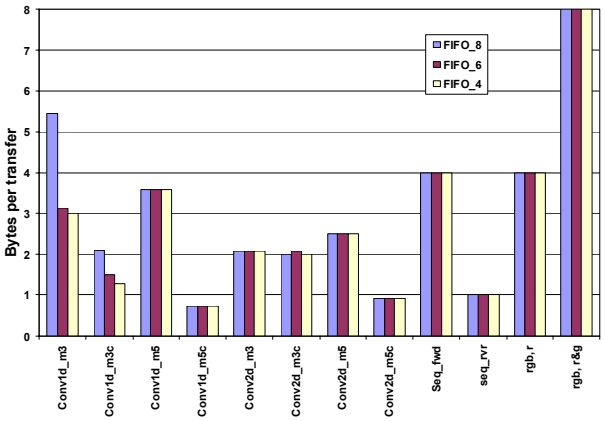

Figure 5. Effective bandwidth in bytes per transfer

As described in Section 5, the simulations are performed over 1D and 2D filters, in row-wise and column-wise traversals, and different filter mask sizes. We also varied the number of FIFO buffers in the DMA unit that stores stream elements to measure their impact on data transfers. An output stream to write processed data is included in all simulations. The last set of results (*rgb,r&g*) is a merged stream descriptor for comparison against the baseline result (*rgb,r*). This represents the access patterns for a set of color pixels in a frame buffer that has interlaced red, green, and blue pixels. In this case, the baseline access pattern fetches only red pixels, while the merged stream descriptor fetches both red and green ones. For example, one HA would request red pixels to filter and identify regions of interest for red objects such as a traffic sign. Another HA would request for green pixels to perform an averaging function, typically found in an autofocus algorithm. The merged stream descriptor describes a new stream record in which the access

overlaps, requiring the DMA unit to fetch two bytes (red, green) in the three color pixel data structure.

In general, the performance of the bus and memory controller is very low, thereby reducing the percentage of useful bandwidth for the DMA unit to approximately 10%, with a read request per 29 cycles and write request per 14 cycles. A bus protocol that supports multiple accesses per requestor, together with a memory controller that can reorder accesses [8], will allow the DMA unit to transfer data more efficiently. However, this limitation only sets the baseline performance as a relative comparison point. Unlike traditional caches that store entire cache lines, the DMA unit stores data based on access patterns and can rely on locality of the stream record access with appropriately sized buffers. Furthermore, data prefetching and buffering allow for effective bandwidth that is higher than normal bus width.

For column traversals, there are opportunities to improve bandwidth utilization with specialized buffers that store a tile of data; that is, the shape of the buffer is configured based on the stream record shape. The baseline (*rgb,r)* result is similar to the *seq_fwd* result as the two share common access patterns despite having different stride values. The behavior of the output streams is also similar to *seq_fwd*. As expected, the performance of the merge output (*rgb,r&g*) is effectively twice the baseline (*rgb,r*) because the bus width and FIFO buffers are larger than the record size, allowing the stream descriptors to properly capture a merged transfer for the DMA unit. Although not shown, the access patterns for different color planes (e.g. red & blue) in the interlaced frame buffer would observe the same effective bandwidth.

Figure 6 shows an example of merged stream descriptors and their associated bandwidth savings. As shown in Figure 6a, two HAs are requesting simultaneously a 3x3 image block in row traversal. Their stream descriptors are then merged into a new data stream. Depending on the operating phase of the two HAs, the stream record requests could overlap completely during execution. This would provide maximum bandwidth improvements, similar to the result of (*rgb,r&g*) presented earlier in Figure 5. However, when the computation is off-phase in terms of accessing data, the DMA unit can fetch a larger block depending on the amount of overlap. The measured data, shown in Figure 6b, indicate that even when the stream records are apart by several columns, there are still bandwidth improvements. This is due in part to the bus width being larger than the width of the stream record. Furthermore, the eight FIFO buffers used in this example are able to store stream elements for later use by either HA. The compiler can schedule execution of the kernels on each HA and use merged stream descriptors to describe the overlap access.

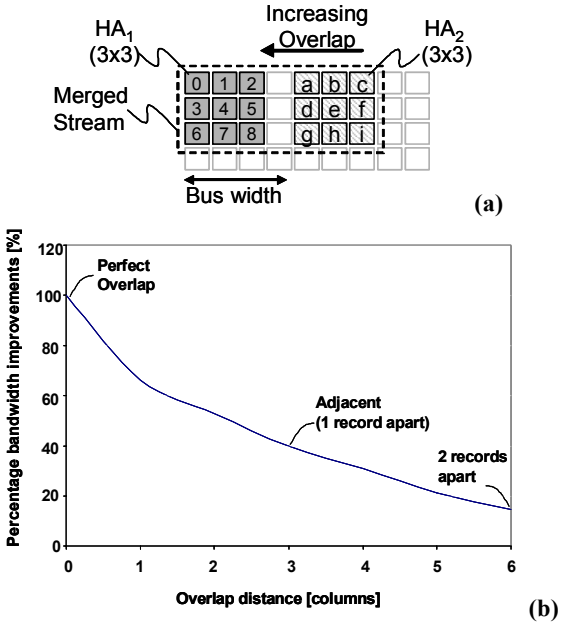

Figure 6. (a) Overlapping stream record access, and (b) Bandwidth improvements of merged stream access

With stream descriptors, the memory access patterns are decoupled from the computation, allowing pre-fetching of ordered data in parallel with computation. This makes the system architecture susceptible to average access latency rather than instantaneous latency. In comparison, standard cache accesses are normally bursty and less tolerant to large latencies of slow memories. To compensate for the cache characteristics, microarchitecture designs have included specialized buffers in caches and memory controller to handle data streams [8, 21]. Alternatively, new processor designs avoid the collusion of stream data with cache traffic with tightly coupled DMA units. This paper advocates the use of the stream programming model and stream descriptors to efficiently schedule data movement on these DMA units. Furthermore, the use of stream descriptors can be applied to other devices [12,13] improving data movement throughout the system.

7. Conclusions and Future Work

This paper illustrates the use of stream descriptors to express memory accesses patterns while disassociating the computation from data movement. The stream descriptors are then manipulated by the compiler to improve a desired target performance metric. Depending on the amount of overlap and buffer

size, merging of data streams using stream descriptor manipulations can reduce bandwidth significantly.

The memory performance results in this paper can be improved with further research in the stream descriptors. First, the API of the stream descriptors can be enhanced with experiments on applications that use traditional data structures such as linked-lists and hash tables. This can lead to more standardized stream descriptors beyond those shown in the paper. Second, the level of compiler interaction with the DMA unit and with the rest of the system can be explored. Complex interconnect hierarchies that include several streaming peripherals, DMA units, and conventional I/O units can be constructed to find the effects of a system-wide management of streaming data.

It is also worthwhile to explore the implications of the stream descriptors on the stream programming model and on hardware implementations. In particular, research on streaming language modifications and the impact on its associated compiler can uncover new methods to better orchestrate stream data movements. The issue of compile time versus run time stream descriptors can also be explored. This paper assumed that the data access patterns could be resolved at compile time, but there are applications that change access patterns at run-time. We have explored several techniques that handle dynamic streams, but that work needs to be formalized.

8. References

[1] Saman P. Amarasinghe, Bill Thies, "Architectures, Languages, and Compilers for the Streaming Domain," *Parallel Architectures and Compilation Techniques (PACT) 2003 Tutorial*

[2] William Thies, Michal Karczmarek, Saman Amarasinghe, "StreamIt: A Language for Streaming Applications," *Proceedings of the 2002 International Conference on Compiler Construction*, pp. 179-195, April 2002

[3] Ian Buck, "Current Brook Specification (0.2)," October 2003. http://merrimac.stanford.edu/brook/brookspec-v0.2.pdf

[4] Ujval Kapasi, et. al., "Stream Scheduling," *Proceedings of the 3rd Workshop on Media and Streaming Processors*, pp. 101-106, December 2001, Austin, TX

[5] William J. Dally, et. al., "Merrimac: Supercomputing with Streams", *SC2003*, pp. 35-43, November 2003, Phoenix, Arizona

[6] Eylon Caspi, et. al. "A Streaming Multi-Threaded Model," *Workshop on Media and Stream Processors*, December 2001

[7] K. Sankaralingam, et. al., "Exploiting ILP, TLP, and DLP with the Polymorphous TRIPS Architecture" *Proceedings of the 30th Annual International Symposium on Computer Architecture*, pp. 422–433, June 2003

[8] S. A. McKee, et. al., "Dynamic Access Ordering for Streamed Computations," *IEEE Transactions on Computers*, Vol. 49, No. 11, pp. 1255-1271, November 2000

[9] A. W. Lim, S. W. Liao and M. S. Lam, "Blocking and Array Contraction Across Arbitrarily Nested Loops Using Affine Partitioning," *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, June 2001.

[10] Pierre Boulet, et. al., "Loop parallelization algorithms: From parallelism extraction to code generation" *Parallel Computing*, vol.24, no. 3-4, pp. 421-444, 1998

[11] Sek Chai, et. al.,"Streaming Processors for Next Generation Mobile Imaging Applications," *IEEE Communications Magazine*, vol.43, no.12, pp. 81-89, Dec 2005

[12] Sek Chai, Abelardo López-Lagunas, "Streaming I/O for Imaging Applications," *International Workshop on Computer Architecture for Machine Perception*, July 2005, pp. 178-183

[13] Abelardo López-Lagunas, Sek Chai, "Memory Bandwidth Optimization through Stream Descriptors," *MEmory performance: DEaling with Applications, systems and architecture* (*MEDEA)*, September 2005

[14] P. Havlak, K. Kennedy, "An implementation of interprocedural bounded regular section analysis", IEEE *Transactions on Parallel and Distributed Systems*, vol. 2, no. 3, pp. 350-360, July 1991

[15] J. Marathe, et. al., "METRIC: Tracking down inefficiencies in the memory hierarchy via binary rewriting", *International Symposium on Code Generation and Optimization,* pp. 289-300, March 2003

[16] Nuwan Jayasena, William J. Dally, "Streams and Vectors: A Memory System Perspective", *Workshop on Media and Stream Processing*, December 2004

[17] Sek Chai, Nikos Bellas, Malcolm Dwyer, Dan Linzmeier, "Stream Memory Subsystem in Reconfigurable Platforms," *Workshop on Architecture Research on FPGA Platforms (WARFP)*, Austin, TX, Feb 2006, 4 pages.

[18] Rudolf Usselmann, "Memory Controller IP Core", January 2002, www.opencores.org

[19] OpenCores Organization, "WISHBONE System-on-Chip (SoC) Interconnection Architecture for Portable IP Cores", revision B.3, September 2002, www.opencores.org

[20] Micron Technology Inc, "Synchonous SDRAM", *Data Sheet for MT48LC2M32B2*, January 2002, www.micron.com/dramds

[21] Subbarao Palacharla, R.E. Kessler, "Evaluating Stream Buffers as a Secondary Cache Replacement", *Proceedings of 21st Annual International Symposium on Computer Architecture*, pp. 24-33, April 1994