Stream Register Files with Indexed Access

Nuwan Jayasena, Mattan Erez, Jung Ho Ahn, and William J. Dally
Computer Systems Laboratory, Stanford University, Stanford, CA 94305, USA
{jayasena,mattan.erez,gajh,billd}@stanford.edu

Abstract

Many current programmable architectures designed to exploit data parallelism require computation to be structured to operate on sequentially accessed vectors or streams of data. Applications with less regular data access patterns perform sub-optimally on such architectures. This paper presents a register file for streams (SRF) that allows arbitrary, indexed accesses. Compared to sequential SRF access, indexed access captures more temporal locality, reduces data replication in the SRF, and provides efficient support for certain types of complex access patterns. Our simulations show that indexed SRF access provides speedups of 1.03x to 4.1x and memory bandwidth reductions of up to 95% over sequential SRF access for a set of benchmarks representative of data-parallel applications with irregular accesses. Indexed SRF access also provides greater speedups than caches for a number of application classes despite significantly lower hardware costs. The area overhead of our indexed SRF implementation is 11%-22% over a sequentially accessed SRF, which corresponds to a modest 1.5%-3% increase in the total die area of a typical stream processor.

1. Introduction

Circuit technology scaling has enabled chips with 10s to 100s of millions of transistors. In the domain of general-purpose microprocessors, translating this raw potential into efficiently utilized compute resources has been a challenge as an increasing portion of chip resources is spent on discovering instruction-level parallelism (ILP) and reducing average memory access latency via cache hierarchies. However, by exploiting data parallelism available in some application domains (e.g. graphics), some application-specific designs have achieved very high compute rates. Data parallelism allows these designs to avoid dynamic ILP extraction and tolerate long memory latencies. Each operation is applied to many data elements over a number of cycles, exposing parallelism and eliminating the need for expensive cache hierarchies. This form of parallelism is exhibited by many increasingly important classes of applications such as multimedia, graphics, cryptography, and scientific simulations.

There has been significant interest in exploiting data parallelism in programmable processors as well. Many general-purpose architectures have added support for parallel processing of sub-word data types [1, 2, 3, 4]. Vector processing [5], which has traditionally been used in high-performance computers with expensive, high bandwidth memory systems, has been applied to microprocessors [6, 7, 8, 9]. Additionally, stream processors exploit data and instruction-level parallelism to achieve high performance for media, signal processing, graphics, and scientific applications [10, 11, 12].

While sub-word parallelism is a limited extension of general-purpose architectures, vector and stream processors provide extensive support for exploiting data parallelism. They also provide storage hierarchies designed to tolerate long memory latencies and supply the high operand bandwidth necessary to perform a large number of operations in parallel. A key part of these storage hierarchies is the *vector* or *stream register file* (VRF or SRF) – a software-managed on-chip storage structure for sequences of data words or records. These register files capture data reuse between operations and provide latency tolerance by allowing large transfers to and from memory to be overlapped with computation on data already in the V/SRF. Current implementations of both these register file types, however, restrict accesses to vectors and streams to be sequential (or some small number of predetermined access patterns).

Several sub-classes of data-parallel application domains such as 2D and 3D signal processing, cryptography, and scientific computing exhibit data reuse patterns that are not amenable to repeated accesses to long vectors or streams in a single order. The sequential access restriction of V/SRFs presents an artificial impediment to extending vector or stream programming models to these applications that are otherwise promising candidates.

This paper explores the use of non-sequential, explicitly indexed access in stream register files[1] in order to enable efficient execution of new classes of applications on stream processors. The key performance benefits of indexed SRF access come from three main sources:

Capture more temporal locality: Allowing arbitrarily ordered accesses to data in the SRF permits a wider range

This work was supported in part by Stanford Graduate Fellowships, Department of Energy under the ASCI Alliances program, and DARPA contracts MDA904-98-C-A933 and NBCH3039003.

[1] We will focus on stream register files, but the indexing technique presented is applicable to vector register files as well.

of data reuse to be captured compared to sequential access, improving performance of memory-bound applications.

Reduced data replication: Repeated access to the same data within a stream requires that data to be replicated in a sequentially accessed SRF. Indexed access allows repeated reference to a single copy of the data, reducing capacity pressure in the SRF. For some applications with large data sets that need to be partitioned in order for the working set to fit in the SRF, this results in a smaller number of larger partitions, reducing associated overheads.

Efficient support for conditional and complex accesses: SRF address computation provides a powerful technique for expressing conditional and complex access patterns at a fine granularity. In addition, memory accesses can be ordered to optimize DRAM throughput and perform complex permutations via SRF indexing.

Indexed SRF access also yields significant memory bandwidth savings over a sequentially accessed SRF – an important benefit as processing power continues to outpace DRAM bandwidth improvements. Our simulations show that indexed SRF access reduces memory bandwidth demands by up to 95% and provides speedups of 1.03x to 4.1x over a sequentially accessed SRF for a sampling of benchmarks with irregular access patterns. In addition, we show that by leveraging the banked nature of large SRAMs, indexed SRF access can be supported with only 11%-22% increase in SRF area. This increase translates to less than 3% increase in the total die area of a typical stream processor based on measurements reported in [13].

The remainder of this paper explores indexed SRF access in more detail. Section 2 provides an overview of stream processing. Section 3 develops a taxonomy of temporal locality in stream programs based on access order and qualitatively discusses how indexed SRF access helps capture more of the available data reuse. Section 3 also describes several common application constructs that benefit from indexed SRF access. Section 4 describes a low-overhead implementation of indexing support in SRFs. Section 5 presents and discusses results of benchmark simulations and parameter sensitivity studies. Section 6 discusses related work, and section 7 concludes.

2. Stream Programming

The stream programming model expresses an application as a collection of *streams* (i.e. sequences of data records) passing through a series of computational *kernels*. For example, Figure 1 shows how a Fast Fourier Transform (FFT) computation passes streams of samples through a series of kernels, each of which performs the butterfly computation for one stage of the FFT. This stream representation allows the programmer to express both parallelism and multiple levels of locality [10].

In stream processors, *kernel locality* is exploited by passing a kernel's intermediate values directly from one ALU to a local register file associated with another ALU without writing them to the SRF. In the FFT example, all the intermediate values generated by the multiplies and adds of a single stage of the butterfly computation are passed directly between ALUs without generating any SRF traffic. Only the inputs and outputs of a butterfly computation are read from or written to the SRF.

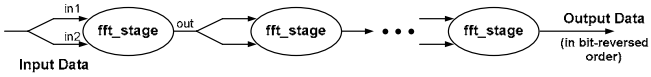


Figure 1: Streaming computation of FFT

At a higher level, *producer-consumer locality* is exploited by forwarding the stream produced by one kernel to the next kernel via the SRF without reading or writing the stream to memory. In the FFT computation, the stream of intermediate results produced by one stage of the FFT is stored in the SRF and then fed as two input streams to the kernel that performs the next stage. Applications are *strip-mined* [14] to ensure that all of the intermediate streams fit in the SRF so that loads and stores to memory needed to spill streams are minimized.

The stream model expresses data parallelism by specifying operations on many records of a stream at once. In the FFT example, a kernel can be applied to all of the elements of its input streams in parallel. Many stream and vector processors exploit this type of data parallelism by operating multiple *clusters* (or vector *pipes*) of ALUs in parallel on different stream elements. In a stream processor, ILP is exploited by performing multiple operations in parallel on each stream element within a cluster under microcode control [10]. Each cluster is associated with a bank of the SRF as shown in Figure 2, providing high bandwidth access to the data in that bank. We will refer to the combination of an ALU cluster and its associated SRF bank as a *lane* of the stream processor.

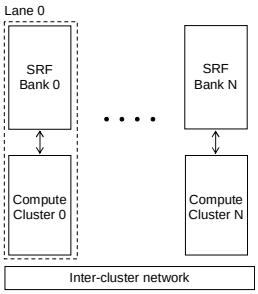


Figure 2: Block diagram of N-lane stream processor

Vector and stream processors hide latency by performing memory operations in parallel with compute kernels. Streams are loaded into the SRF while kernels read earlier streams from the SRF and generate outputs to the SRF while still earlier results are stored back to memory. A single instruction loads or stores an entire stream, and therefore, a handful of instructions are sufficient to launch enough accesses to cover very long memory latencies.

To handle irregular data structures, stream memory operations may be indexed. Indexed loads or *gather* operations collect data from arbitrary locations in memory into a sequential stream. Similarly, indexed stores or *scatter* operations store the elements of a sequential stream to arbitrary memory locations. In a stream or vector processor without an indexed SRF, gather and scatter operations are often used to reorder data through memory.

3. Indexed SRF Access

In this section we broadly classify the types of stream locality and identify some common application constructs that benefit from indexed SRF access.

3.1. Stream Locality

Temporal locality available in applications at the stream level can be broadly categorized based on the ordering of repeated accesses as follows:

In-order stream locality: Multiple accesses to a stream in the same order by one or more kernels.

Reordered stream locality: Multiple accesses to a stream in different order(s) by one or more kernels.

Intra-stream locality: Repeated access to subsets of data within a stream during a single kernel's execution.

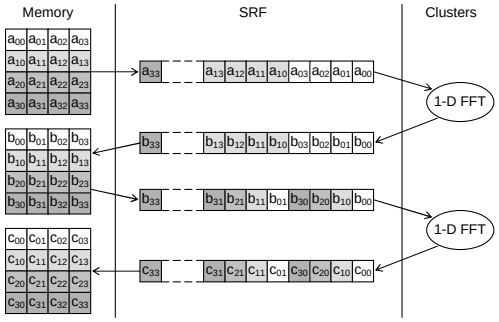
Strictly sequentially accessed SRFs only exploit in-order stream locality. Indexed SRF access enables reordered reuse to be captured by allowing streams to be read out in the desired new order directly from the SRF, and intra-stream reuse to be captured by repeatedly referring to a single copy of the data.

3.2. Application Constructs

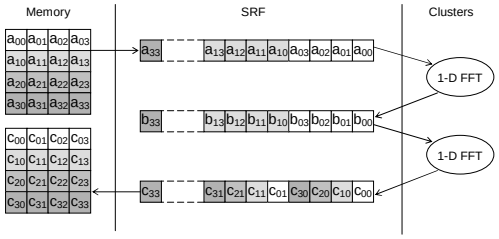
Indexed SRF access enables efficient support for several constructs common in applications that are amenable to stream programming but perform poorly with sequential SRF accesses. A few of these are summarized below.

Multi-dimensional array accesses: Accesses along different dimensions of a multi-dimensional array can be supported efficiently by capturing reordered stream locality in the SRF. Many signal processing applications with two- or higher-dimensional data sets require such operations. For example, consider the simplified 2D FFT shown in Figure 3. The first invocation of the 1D FFT kernel generates an intermediate result array in row-major order. The second kernel invocation needs to apply the algorithm along the columns, requiring a reordering of the array. This requires writing the entire array to memory and re-reading it in column-major order if only sequential SRF accesses are allowed. However, indexed SRF access allows the column-major access to take place directly from the SRF.

Neighbor accesses in multi-dimensional arrays: Accessing data neighboring a given element in a multi-dimensional data structure using sequential accesses requires adjacent values in all dimensions to be contiguous within the stream. A simple 2D example of such a case is shown in Figure 4, where a 3x3 filter kernel is applied to the row shown in gray. Applying the filter to subsequent rows requires reordering the data set. Avoiding this overhead without SRF indexing requires complex state management in the filter kernel with the use of local scratchpad memories.[2] Indexed SRF access makes it trivial to perform the neighbor accesses by capturing the reordered reuse. Operations such as these are common in some classes of scientific simulations with regular grid structures and in various filters and solvers.



(a) Sequential SRF requires reordering through memory



(b) Indexed SRF allows reordered access in SRF

Figure 3: Data reordering for 2D FFT

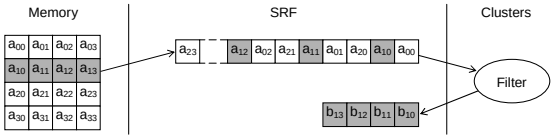
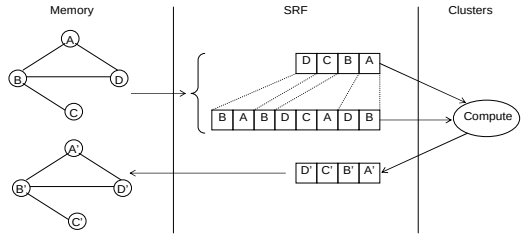


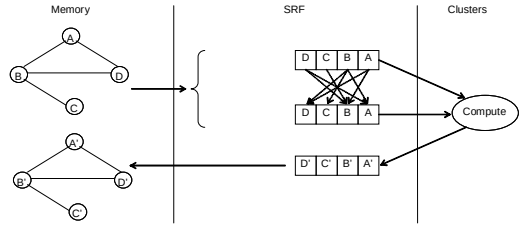
Figure 4: Neighbor accesses in 2D array

Neighbor accesses in irregular graphs: Accessing neighbors of nodes in irregular graph structures using sequential streams result in nodes that neighbor multiple other nodes being replicated as shown in Figure 5. Exploiting this intra-stream locality using indexed SRF access reduces memory traffic and the space occupied in the SRF. Application classes with such accesses include scientific applications with irregular structures and some graph traversal algorithms.

[2] Small 2D neighborhood accesses such as this simple example can be implemented more efficiently by treating each row as a separate stream. However, that technique does not scale to higher dimensions or larger neighborhoods due to the limited number of streams supported in hardware, and therefore is not considered here.



(a) Sequential SRF requires repeatedly accessed records to be replicated



(b) Indexed SRF allows repeated access to a single copy

Figure 5: Neighbor accesses in irregular graph

Table lookups: Data-dependant table lookups are used in a variety of applications for algorithmic reasons as well as for storing pre-computed values as an optimization. With a sequential-only SRF, these accesses require indexed gather operations from memory as described in section 2, which are inefficient if the amount of computation performed per lookup is small. With indexed SRF access, lookups can be performed in the SRF if the table or a useful partition of it can fit in the SRF. An alternate method of supporting table lookups is to incorporate a scratchpad memory in addition to the SRF. While architectures such as [10] incorporate a small scratchpad memory, increasing its size to be a significant fraction of the SRF, particularly with comparable bandwidth, introduces high overheads while benefiting only a limited set of applications.

Conditional accesses: Conditional control flow is an expensive operation in stream and vector processors and is often translated into conditional data accesses in order to improve efficiency [15, 16]. Conditional stream accesses without SRF indexing require communication among lanes as data statically mapped to SRF banks must be accessed sequentially and distributed among clusters based on dynamically evaluated conditions. In some cases, however, conditional computation of SRF indices provides an alternative for expressing conditionals without the overhead of cross-lane communication, as in the sort benchmark described in section 5.2.

4. Implementation

A key advantage of a sequentially accessed SRF is the ability to provide high bandwidth using single-ported memories by accessing wide blocks of data at a time since the access order is known to be sequential. The following subsections describe the implementation of a conventional SRF and a technique for supporting indexed access while retaining high bandwidth and energy efficiency for sequential accesses.

4.1. Conventional SRF SRAM

The SRAM array of a conventional SRF is optimized for sequential block accesses. In an *N*-lane stream processor, each SRF access reads or writes *N* x *m* logically contiguous words where *m* is the number of words accessed in each lane. While this conceptually forms a single wide memory array stretching across all lanes, such large SRAMs are typically implemented as several smaller arrays due to access time and energy considerations [17]. An efficient implementation of a conventional 128KB SRF with word size = 32b, $N = 8$, and $m = 4$ is shown in Figure 6. Each bank of the SRF is implemented as a separate 16KB SRAM. Since all banks of the SRF access the same row at once, a single row decoder is shared for area efficiency. Internally, each bank is composed of $s = 4$ sub-arrays of 4KB each and a hierarchical bitline structure.

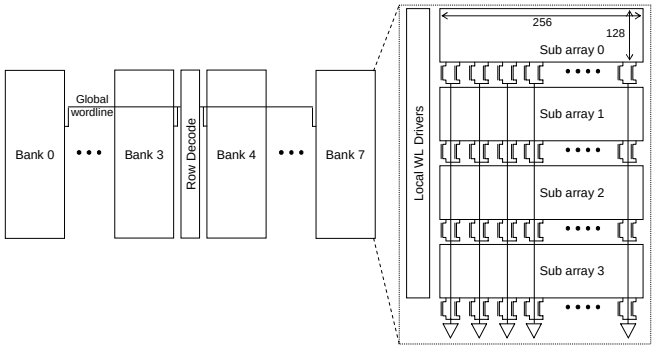


Figure 6: 128KB sequential SRF SRAM (N=8, m=4, s=4)

4.2. Indexed SRF SRAM

Each lane may access a different row of the SRAM in an indexed SRF, requiring a dedicated row decoder for each bank. We will refer to such an SRF configuration as *ISRF1*, indicating that each lane may perform one indexed access per cycle. However, indexed accesses are often performed at word granularity, and limiting each lane to a single access per cycle reduces the bandwidth for indexed accesses by a factor of *m* compared to sequential streams. The existing sub-array structure within each bank can be used to increase the bandwidth of indexed accesses by performing an independent one-word access in each sub-array. This provides a peak bandwidth of *s* words per cycle per lane if no two accesses are to the same sub-array.

Figure 7 shows the modifications necessary to support indexed access, including a detailed view of a single SRF bank that supports up to 4 independent, 1-word accesses in parallel. We will refer to this configuration as *ISRF4*. The

key modifications are an independent row decoder and an additional 8:1 column multiplexer at each sub-array. Outputs of the 8:1 multiplexers are interleaved among the global bitlines such that accesses from multiple sub-arrays do not cause conflicts on the bitlines. Sequential 4-word accesses are still provided from a single sub-array bypassing the 8:1 multiplexer, thereby retaining the energy efficiency of sequential SRF accesses. This structure allows an SRF bank to perform either a single 4-word access for sequential streams, or up to 4 one-word accesses for indexed streams using only single-ported SRAMs and one set of global bitlines.

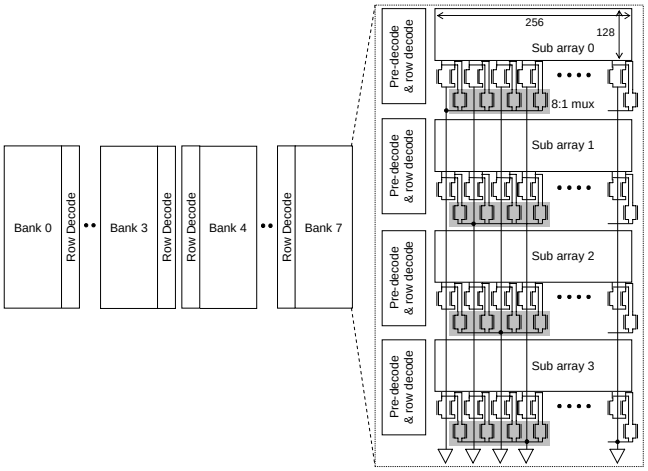


Figure 7: 128KB indexed SRF SRAM (N=8, m=4, s=4)

4.3. Conventional SRF Access Mechanism

Although the SRF is single-ported, applications require access to multiple streams simultaneously. Therefore, the SRF port is time-multiplexed over several streams, and *stream buffers* are used to match the access characteristics of the SRF to that of the computational kernels as shown in Figure 8(a). On each SRF access, *N* x *m* words are transferred to/from the SRF from/to a single stream buffer. The compute clusters, on the other hand, access the stream buffers *N* words at a time (one word per cluster), but may access multiple stream buffers at once. A separate set of stream buffers are used to mediate transfers between the SRF and the memory system.

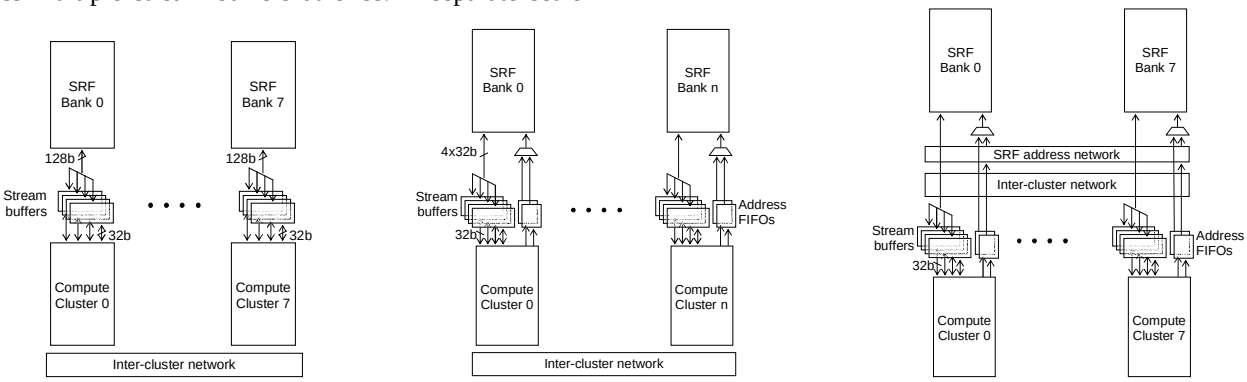
Address generation for the SRF is performed by counters that keep track of the next block to be accessed for each stream. Arbitration among streams for access to the SRF port is dynamic and decoupled from kernel execution.

4.4. Indexed SRF Access Mechanism

For indexed access, each cluster generates a sequence of addresses for each indexed stream. Each of these address sequences has a dedicated address FIFO as shown in Figure 8(b). Each address FIFO is associated with a stream buffer that holds the data for the indexed accesses and provides the same interface to the compute clusters as sequential streams. In our implementation, the address FIFOs hold record addresses generated using the ALUs in the compute clusters, reducing the need for specialized address generation hardware. Counters at the head of the FIFOs break up each record access into a sequence of single-word indexed accesses, significantly reducing the address generation overhead imposed on the compute clusters.

Arbitration for the SRF port is implemented as a two-stage process. During the first stage, global arbitration selects a sequential stream or all indexed streams. If indexed streams are chosen, the second stage, performed locally in each lane, determines which indexed accesses take place. The second stage is performed speculatively in parallel with the first in order to reduce SRF access latency.

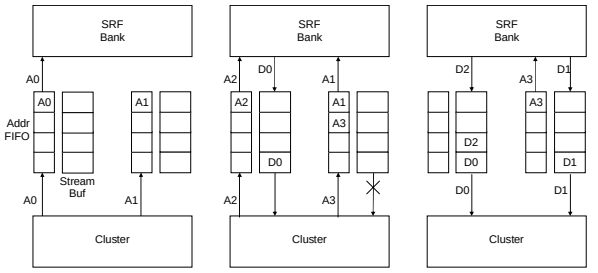
Figure 9 shows an example sequence of indexed reads involving two indexed access streams. For simplicity, only a single lane and the two streams used in the example are shown. Figure 9(a) shows that in cycle *i*, the cluster issues one access each to the two streams by placing addresses in the address FIFOs. During local arbitration, both accesses A0 and A1 are determined to be to the same sub-array and are serialized. The delayed A1 access is performed in cycle *i+1* along with other new or pending non-conflicting accesses as shown in Figure 9(b). The cluster stalls when it tries to read the data for the accesses issued in cycle *i* as the data for access A1 is not available yet due to the conflict



(a) Sequential SRF access **(b) Indexed SRF access** **(c) Cross-lane indexed SRF access**

Figure 8: SRF access mechanisms (N=8, m=4, s=4)

with A0. Figure 9(c) shows the cluster data access succeeding after the data for both streams is available. Indexed SRF access is shown as a single cycle operation in this example for brevity. In reality, it is a pipelined multi-cycle operation.



(a) cycle i **(b) cycle (i+1)** **(c) cycle (i+2)**

Figure 9: Indexed SRF access example

4.5. Cross-Lane Indexed Accesses

The mechanism described in section 4.4 limits indexed access from a cluster to the SRF bank in its lane. This level of indexing, coupled with statically scheduled inter-cluster transfers, is sufficient to support data reordering patterns that can be analyzed at compile-time. However, it is inefficient for data-dependant access patterns since any static schedule must allocate communication resources assuming worst-case conflicts. Cross-lane indexed access allows any cluster to access any SRF bank with dynamic conflict resolution. However, this requires communicating indices and data among lanes. The existing inter-cluster network is modified to accommodate the data transfers for cross-lane SRF access and a dedicated network is added for index communication as shown in Figure 8(c). In our implementation, both these networks are fully connected crossbars similar to the inter-cluster network of the Imagine processor [10]. Explicit inter-cluster communications receive higher priority when resolving conflicts with cross-lane SRF accesses since those operations are statically scheduled and assume a fixed latency to completion.

4.6. Overheads

Area overheads for the SRF designs with indexed access were estimated using a modified version of the Cacti 3.0 [18] models and custom floorplans. ISRF1 and ISRF4 configurations incur 11% and 18% area overheads over a sequential-only SRF of equal capacity. Much of the extra overhead of ISRF4 over ISRF1 is in the additional address busses and per-sub-array predecoders. Area overhead of cross-lane indexing is 22% over a sequential SRF, with much of the incremental overhead over ISRF4 associated with the address network. The above overheads are as a fraction of SRF area only and represent 1.5% to 3% of overall die area based on the Imagine processor statistics reported in [13].

SRF energy consumption for sequential stream accesses is comparable for both the indexed and sequential-only designs. Indexed single-word accesses in our design consume approximately 4x the energy per word in the SRAM array compared to sequential stream accesses due to increased column multiplexing. However, the estimated energy consumed by an indexed SRF access at approximately 0.1nJ in a 0.13µm technology is still an order of magnitude lower than the ~5 nJ required for an off-chip DRAM access.

4.7. Programmer Interface

We have added indexed SRF support to the *KernelC* language and compiler developed for the Imagine stream processor [19]. KernelC is a subset of C with added stream and communication semantics. Figure 10 shows a simple kernel that performs a table lookup via SRF indexing. Indices into streams are specified using syntax similar to C array notation as shown in line 8. Table 1 lists the types of indexed SRF streams supported. Currently we do not support cross-lane indexed write streams.

```
1   kernel lookup(
2       istream<int> in,      // sequential in stream
3       idxl_istream<int> LUT,// indexed in stream
4       ostream<int> out) {   // seq. out stream
5     int a, b, c;
6     while(!eos(in)) {
7       in >> a;       // sequential stream access
8       LUT[a] >> b;   // indexed stream access
9       c = foo(a, b);
10      out << c;
11    }
12  }
```

Figure 10: Indexed SRF access code example

Access type	Stream type	Access syntax
In-lane read	`idxl_istream<type>`	`strm[idx] >> a;`
In-lane write	`idxl_ostream<type>`	`strm[idx] << a;`
Cross-lane read	`idx_istream<type>`	`strm[idx] >> a;`

Table 1: Indexed SRF access and stream types

Tolerating the non-deterministic latency of SRF reads, which varies due to arbitration for the SRF port and array/bank conflicts, is a key concern in indexed access. As seen in the example of Figure 9, attempting to read the data before the access completes leads to processor stalls. In order to minimize the probability of stalls, each access is split into separate address issue and data read operations during compilation and are scheduled several cycles apart.

5. Evaluation

We compare the performance of a collection of application and synthetic benchmarks on four machine configurations in order to evaluate the impact of indexed SRF access. Table 2 summarizes the different machine configurations, and Table 3 lists their key parameters.

Config.	Description
Base	Sequential SRF backed by off-chip DRAM, similar to the description in section 2.
ISRF1	Indexed SRF with one word per cycle per lane in-lane indexed bandwidth (no sub-banking) & cross-lane indexing. Backed by off-chip DRAM.
ISRF4	Indexed SRF with up to 4 words per cycle per lane in-lane indexed bandwidth (4 sub-arrays per lane) & cross-lane indexing. Backed by off-chip DRAM.
Cache	Sequential SRF backed by on-chip cache and off-chip DRAM.

Table 2: Machine configuration summary

Parameter	Base	ISRF1	ISRF4	Cache
Lanes	8			
System clock	1 GHz			
Peak compute	32 GFLOPs			
Peak DRAM bandwidth	9.14 GB/s			
SRF capacity	128 KB			
Peak sequential SRF bandwidth	32 words/cycle (128 GB/s)			
Sequential SRF latency	3 cycles			
Stream buffer size (per lane per stream)	8 words			
Address FIFO size (per lane per stream)	n/a	8	8	n/a
Peak in-lane indexed SRF bandwidth (words/cycle/cluster)	n/a	1	4	n/a
Peak cross-lane indexed SRF bandwidth (words/cycle/cluster)	n/a	1	1	n/a
In-lane indexed SRF latency (cycles)	n/a	4	4	n/a
Cross-lane indexed SRF latency (cycles)	n/a	6	6	n/a
Cache size (KB)	n/a	n/a	n/a	128
Cache associativity	n/a	n/a	n/a	4
Cache banks	n/a	n/a	n/a	4
Peak cache bandwidth (GB/s)	n/a	n/a	n/a	16
Cache line size (words)	n/a	n/a	n/a	2
Cache replacement policy	n/a	n/a	n/a	LRU

Table 3: Machine parameters (SRF access latencies assume no arbitration failures or bank conflicts)

Base, *ISRF1*, and *ISRF4* configurations are similar to the designs described in this paper so far. *Cache* configuration is similar to the cache memories that have been incorporated into several vector architectures as an alternative technique for capturing temporal locality [20, 21], and is provided for comparison purposes. Note that the cache stores redundant copies of data in the SRF and incurs a 100%-150% area overhead over a sequentially accessed SRF. Cache parameters such as the short line size are based on previously published vector cache studies [22, 23]. Caching is only performed for streams with potential for temporal locality in order to minimize cache pollution. All machine configurations assume 4 fully pipelined functional units which support integer and floating-point add and multiply ops, and a single unpipelined divider unit per lane. SRF and cache access is assumed to be fully pipelined.

5.1. Simulation Methodology

All benchmark simulations were performed using a cycle accurate simulator. Benchmark kernels were written in KernelC and scheduled using an automated scheduler based on [19]. Ideally, indexed SRF reads would be scheduled such that addresses are issued as early as possible and data read as late as possible (subject to critical path and buffering constraints) in order to minimize stalls due to bank and array conflicts. However, our scheduler currently does not support variable latency operations, and therefore all benchmarks were scheduled with a fixed address and data separation of 6 cycles for in-lane accesses and 20 cycles for cross-lane accesses. The sensitivity of performance to address and data separation and other parameters is explored in section 5.4.

5.2. Application Benchmarks

To evaluate the performance impact of indexed SRF access, we simulated a set of application benchmarks and a synthetic benchmark representative of data parallel applications with irregular access patterns. The synthetic benchmark was parameterized to rapidly explore a wide range of the application space, which would otherwise require many more applications to be implemented. Note that indexed SRF access does not benefit all applications – particularly those that only require sequential streams. However, as stream programming extends to application classes with complex data access patterns, such as scientific computing and sophisticated signal and media processing algorithms, we believe that this technique will benefit a significant portion of stream applications. The benchmarks are described below.

2D FFT: A 2-dimensional FFT on a 64x64 array. The entire array fits in the SRF. Both the base and indexed SRF implementations perform each of the 1D FFTs along the first dimension across all lanes. The base version then performs a 90º rotation of the data array through memory and applies the same computation along the second dimension. In the indexed SRF version, each cluster performs the second dimension FFT for the data in its local bank of the SRF using in-lane indexed accesses.

Rijndael: A block encryption algorithm that was recently adopted as the Advanced Encryption Standard [24]. While the algorithm can be implemented in many ways, we consider an optimized implementation that relies on large numbers of lookups into pre-computed tables [25]. In the base case, the table lookups generate memory accesses while the indexed case performs the lookups in the SRF. In order to facilitate high lookup bandwidth, the tables are replicated in each lane, enabling in-lane indexed SRF access. Both versions implement the *cipher block chaining* (CBC) mode of the algorithm [26], with each cluster encrypting an independent data stream. Such an implementation is suitable for encrypting network traffic or other applications with many independent data streams.

Sort: Merge sort of 4096 values. Each iteration of the algorithm requires conditional merging of two input

streams, which in a sequential SRF requires the use of conditional streams [16], resulting in cross-lane communication on every iteration. With SRF indexing, the conditional inputs are formulated as conditional address computations, and no cross-lane communication is necessary until all data in each lane is internally sorted.

Filter: Application of a 5x5 convolution filter to a 256x256 2D image. The base implementation temporarily stores neighborhood data in scratchpad memories to avoid replication in the SRF while the indexed SRF version simply reads the neighborhood data directly from the SRF.

Irregular Graph Simulation (IG): Synthetic benchmark that simulates neighbor interactions in a static irregular graph. For each node in the graph, all of its neighbors are accessed, and the node value is updated based on the neighbors' values. The graph is assumed to be much larger than the available SRF space, requiring the data set to be partitioned into several strips. No data is replicated across lanes, and therefore, all indexed SRF accesses are cross-lane. Amount of computation per neighbor access, average graph degree, and strip length are parameterized to explore the application space. Table 4 summarizes the parameter values for a cross section of the data sets explored. The three letter suffix at the end of the data set names are as follows: the first letter indicates a "*S*parse" or "*D*ense" graph – an indication of the average number of neighbors per node; the second letter specifies whether the data set is *C*ompute or *M*emory limited on the base architecture; and the third letter indicates *S*hort or *L*ong strip size. The strip sizes for the base and indexed SRF implementations of each data set were set to occupy approximately the same storage space in the SRF.

Data set	FP ops per neighbor	Avg. graph degree	Avg. strip size	
			Base SRF	Indexed SRF
IG_SML	16	4	1163	2316
IG_SCL	51	4	1163	2316
IG_DMS	16	16	265	528
IG_DCS	51	16	265	528

Table 4: Parameters for IG benchmark datasets (strip size is the number of neighbor records processed per kernel invocation)

5.3. Results and Discussion

All results presented in this section assume the benchmarks are executed multiple times in software pipelined loops. This assumption is representative of most applications of interest since they are typically applied repeatedly to long input streams such as sequences of images or multiple strips of large, partitioned data sets.

Figure 11 shows off-chip memory bandwidth requirements of the benchmarks for ISRF and Cache configurations, normalized to the Base case (note that ISRF1 and ISRF4 have identical memory bandwidth requirements). Indexed SRF access provides significant bandwidth savings for all benchmarks except Sort and Filter. 2D FFT benefits by eliminating a data set reordering through memory, and Rijndael benefits by eliminating large numbers of table lookups from memory as the tables fit in the SRF. Reductions in the IG benchmarks for ISRF are due to eliminating intra-strip node and neighbor record replication, partially offset by the overhead of indices (pointers) into the condensed neighbor data array. Sort and Filter do not gain any bandwidth reduction as all available locality is captured by the base configuration as well.

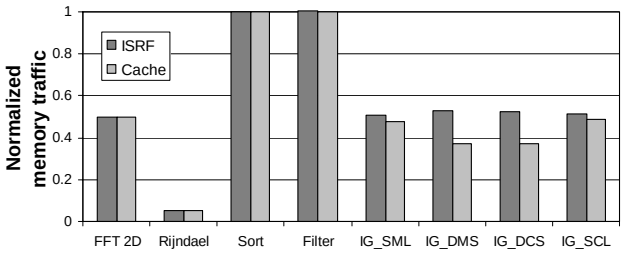


Figure 11: Off-chip memory traffic normalized to Base

Cache also completely captures the FFT 2D data reordering and Rijndael table lookups, achieving bandwidth reductions similar to ISRF for those benchmarks. Cache outperforms ISRF in terms of locality capture for the irregular (IG) benchmarks as it is also able to capture inter-strip reuse when partial overlaps exist between partitions of the data set.

Figure 12 shows the breakdown of execution time of the benchmarks. Our current implementation limits each indexed stream to issuing a single indexed SRF access per cycle for simplicity. Therefore, ISRF1 and ISRF4 differ only for benchmarks with more than one indexed stream. In our benchmark set, only Rijndael and Filter require multiple indexed streams, and therefore, data for ISRF1 is shown only for those benchmarks.

In Figure 12, the *Kernel loop body* component corresponds to the time spent executing the main loops of the kernels where much of the useful computation is performed. *Memory stall* corresponds to time spent waiting for memory or cache accesses to complete. *SRF stall* is time spent stalling for SRF accesses to complete. *Kernel overheads* include time spent executing kernel code before and after the main loop body, including software pipeline fills/drains and the impact of load imbalances among lanes.

As Figure 12 shows, ISRF4 provides speedups over the Base configuration for all benchmarks. FFT 2D and Rijndael on the Base machine are constrained by memory bandwidth, and reducing memory traffic via indexed SRF access provides speedups of 2.24x and 4.11x respectively.

Improvements in the Sort benchmark on ISRF4 are a result of efficient support for conditional SRF access reducing kernel loop execution time. The speedup of the Filter benchmark is due to efficient access of neighbor values directly from the SRF, also reducing kernel loop execution time.

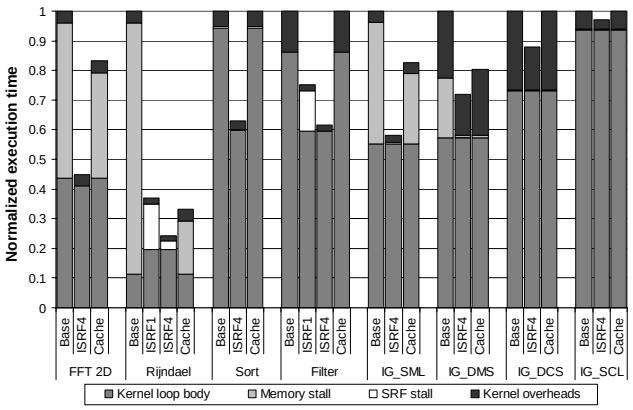


Figure 12: Execution times normalized to Base (ISRF1 shown only where it differs from ISRF4)

The IG benchmarks span a wide range of application characteristics. IG_SML and IG_DMS have low compute density and are constrained by memory bandwidth. Capturing intra-strip locality in these data sets in the SRF improves performance of both. Another factor contributing to improved performance on ISRF4 is the increased strip sizes that can be accommodated in the SRF as a result of eliminating replication, which amortizes kernel start and end overheads over larger batches of useful computation. The two dominant components of these overheads for this benchmark are software pipeline overhead and load imbalance between lanes. While we do implement dynamic load balancing using the technique presented in [16], some imbalance still exists at the end of each strip as all lanes remain occupied until the last lane has competed processing its final input. These overheads represent a significant portion of the run time for IG_DMS and IG_DCS which have shorter strip sizes. Other overheads such as application initialization operations are minimal in this benchmark but may be significant in real applications, which would further benefit ISRF4.

IG_SCL represents a scenario where SRF indexing provides little performance benefit since it is compute limited and has long strips even on the Base configuration.

ISRF4 also outperforms the Cache configuration for all benchmarks. The data set reordering of FFT 2D is fully captured in the cache, but unlike ISRF4, an explicit reordering operation must still be performed on the data in the SRF. Therefore, the software pipelined loop in the Cache configuration is longer than in ISRF4, limiting the performance benefit from the cache as the iteration interval is bounded by the number of data sets that fit in the SRF simultaneously. For Rijndael and IG_SML, the cache captures at least as much locality as ISRF4 but does not have adequate bandwidth to eliminate all memory stalls. The cache does not provide the conditional and complex SRF accesses enabled by ISRF4 that benefit Sort and Filter, and consequently, does not provide any speedup for these benchmarks. The cache also does not eliminate data replication in the SRF, and therefore, does not provide the strip size increases that improve the performance of IG_DMS and IG_DCS on ISRF4.

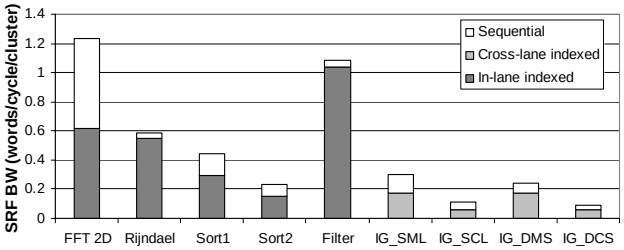


Figure 13: Sustained SRF bandwidth demands

Finally, none of the benchmarks suffer significantly from a lack of indexed SRF bandwidth on ISRF4. On the other hand, Rijndael and Filter spend 42% and 18% of the execution time on SRF stalls on ISRF1, demonstrating the benefit of high indexed SRF bandwidth for benchmarks with large amounts of indexed accesses. Figure 13 shows the sustained SRF bandwidth demands in the main loops of the benchmarks (both Sort1 and Sort2 kernels are used by the Sort benchmark). While the sustained bandwidths are relatively low, the access patterns are bursty, and decoupled early address issue and stream buffers play a critical role in minimizing SRF stalls.

5.4. Parameter Studies

This section evaluates the sensitivity of the design to several hardware and software parameters. Any parameters not explicitly specified or varied in these studies are set to the same values as the ISRF4 configuration described earlier in section 5.

As discussed in section 4.7, issuing SRF addresses for read operations early enough in order for the read to complete before the data is needed is critical for reducing SRF stalls. However, increasing the separation between address and data can extend the static schedule length of kernels leading to a loss of performance. Figure 14 shows the variation of static schedule lengths of the inner loops of benchmark kernels as address and data separation increases. This separation is varied from 2 to 10 cycles for in-lane indexing and from 2 to 24 cycles for cross-lane indexing. IGraph1 is used in IG_SML and IG_DMS benchmarks, and IGraph2 is used in IG_DCS and IG_SCL.

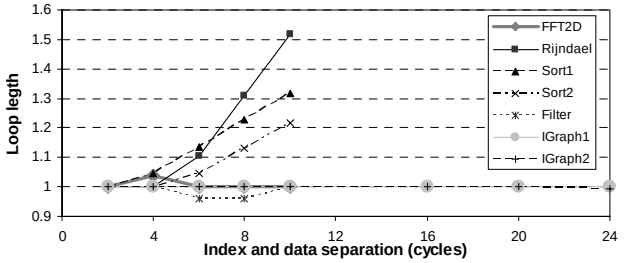


Figure 14: Static schedule length variation of loop bodies

Rijndael, Sort1, and Sort2 kernels have loop-carried dependencies that affect index computation, which causes schedule length to increase rapidly with address and data separation. FFT 2D, Filter, and the IGraph kernels, in contrast, are able to use software pipelining to tolerate very long separations with no increase in static schedule length. The minor fluctuations in schedule lengths are due to randomized algorithms used in the scheduler.

Figure 15 shows the variation in execution time of kernels as address and data separation increases for the in-lane indexed kernels. Performance initially improves for all benchmarks with increasing separation as SRF stalls reduce, and then degrades as schedule length increases dominate. In the case of FFT 2D, which shows no schedule length increase, performance degradation occurs as a result of increased overheads due to deeper software pipelining.

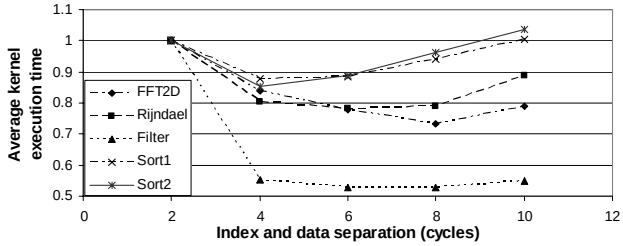


Figure 15: Execution time of in-lane indexed kernels

Figure 16 shows the execution time variation for the cross-lane indexed kernels. These kernels are able to tolerate long address and data separations due to their high compute density and lack of loop-carried dependencies. The irregularity in the curve for IGraph1 at 20 cycles corresponds to an increase in the software pipeline length.

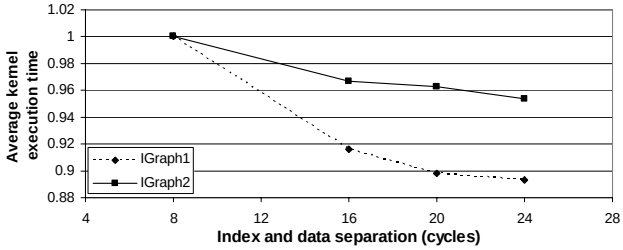


Figure 16: Execution time of cross-lane indexed kernels

Figure 17 shows the variation in sustained in-lane indexed SRF throughput as a function of the number of sub-arrays within a SRF bank and the size of the address FIFOs. These results were obtained using a micro-benchmark that issues 4 random reads per cycle per cluster on every cycle. Throughput increases with FIFO size as more addresses are issued before stalling on conflicts, and with the number of banks as the probability of conflicts declines. However, utilization of available bandwidth decreases as the number of sub-arrays increases due to *head-of-line blocking* (i.e. conflicting accesses stalling subsequent requests). While throughput can be improved by adding an additional FIFO per sub-bank, we do not implement this for design simplicity as current bandwidth is sufficient for application requirements. These experiments were conducted with an 8 cycle separation between address issue and data consumption, and therefore FIFO sizes beyond 8 are not meaningful. Reducing the separation to 4 and 2 cycles reduced throughput by approximately 16% and 50% on average. Arbitration among streams for SRF access was performed using a simple round-robin scheme. Complex arbiters that prioritize streams likely to cause stalls were found to provide less than 10% improvement in throughput.

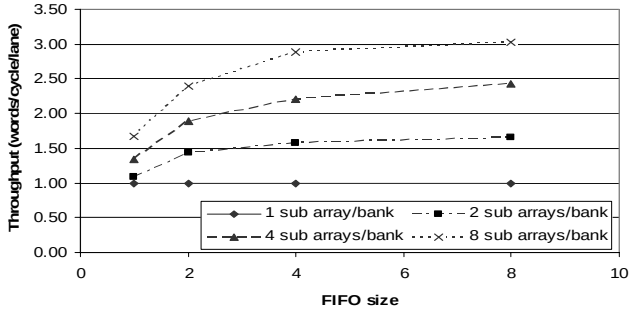


Figure 17: In-lane indexed throughput variation with number of SRF sub-arrays and address FIFO size

Figure 18 shows cross-lane indexed access throughput as a function of the peak number of cross-lane accesses permitted in a bank of the SRF each cycle and the percentage of cycles in the static kernel schedule that contain inter-cluster communications unrelated to cross-lane SRF access. The results in Figure 18 were obtained by issuing 1 random cross-cluster read and 3 sequential stream accesses per cycle per cluster. Increasing the number of network ports per SRF bank from 1 to 2 provides a significant improvement in throughput, while increasing this number beyond 2 provides only marginal improvements. Adding an additional network port on the SRF side alone in the linear lane arrangement of Figure 8(c) does not require an increase of the bisection bandwidth and is thus relatively cheap to implement. However, this is not the case for more complex layouts like the 2D grid of lanes proposed in [27]. Consequently, the results presented in section 5 were obtained using only 1 network port per SRF bank.

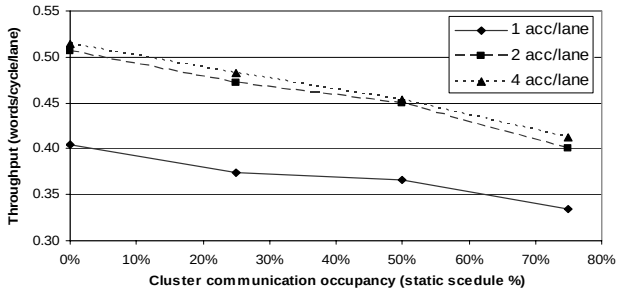


Figure 18: Cross-lane indexed throughput variation with number of SRF accesses per cycle and inter-cluster traffic

Figure 18 also shows that the reduction in cross-lane SRF throughput is 20% or less for a wide range of inter-cluster communication traffic loads. This shows that in the presence of both in-lane and cross-lane SRF traffic, the dominant factor in reducing cross-lane access throughput is contention for SRF access rather than inter-cluster traffic. Therefore, multiplexing both types of inter-lane traffic over a single network instead of two dedicated networks is the preferred design option, particularly given the high area cost of the networks.

6. Related Work

The work in this paper extends the stream register file architecture introduced by the Imagine stream processor [10] and is generally applicable to vector processors as well. Both vector and stream register files rely on wide sequential accesses to supply data to a number of parallel compute clusters.

Asanovic [23] discusses VRF implementation options but does not consider explicitly indexed access. Many vector machines support masked accesses [15], which are a small subset of the patterns enabled by indexed access. Some designs, such as [7, 28], provide support for a small number of specialized vector permutations, usually targeted at specific applications. The CM-5 [29] vector unit provided a subset of the enhancements proposed in this paper. It supported in-lane strided access to the VRF and at most one vector operand per instruction could be accessed in arbitrary order (in-lane) using indirect addressing. MultiTitan [30] used a scalar register file to store vectors, enabling arbitrary access, but did not allow dynamic index computation and is limited to small register file sizes due to the multi-port requirements of scalar registers. Corbal et. al. [31] explore a 2-level VRF for enhancing multi-dimensional accesses but do not address the issues of data reordering and general permutations.

Architectures like Imagine and Cray-2 [32] have implemented indexable SRAMs decoupled from the V/SRF. However, these typically provide much lower bandwidth than the SRF or VRF and lead to fragmentation of total available storage compared to a unified structure. In addition overheads are incurred in transferring data between the indexed memory and the V/SRF. A number of vector architectures incorporate caches between the VRF and off-chip memory [20, 21], similar to the Cache configuration which we compare to our proposed design in section 5. In addition, [23] also presents a few specialized cache organizations for vector processing.

On-chip caches are also common in general purpose CPUs. Data prefetching techniques attempt to achieve the same goals as stream and vector programming by staging data into the cache before it is needed [33]. However, caches do not provide software control over data replacement and are susceptible to conflict misses if the prefetches are not timed perfectly. DSPs provide software managed flat SRAM arrays or caches or both [34, 35]. While these memories can be indexed, they typically provide lower bandwidth than V/SRFs.

A few recent configurable architectures such as Smart Memories [36] and TRIPS [37] also support forms of stream computation. These architectures implement stream storage using configurable memory mats, and while they do support indexed access, are less efficient for sequential stream accesses than dedicated SRF implementations.

7. Conclusions and Future Work

Extending the stream programming model to applications with complex data access patterns is hindered by the strictly sequential access semantics of stream/vector register files. Non-sequential access to data in the V/SRF requires reordering through lower bandwidth levels of the memory system, potentially requiring off-chip accesses. This has a negative impact on the performance of many applications that require such reorderings and makes inefficient use of memory system bandwidth.

We propose a SRF architecture that allows the use of explicit indices to access streams in non-sequential orders, providing the ability to capture a wider range of temporal locality available in applications, reduce data replication in the SRF, and efficiently support fine-grained conditionals and complex access patterns. We showed that several common application constructs such as table lookups, multi-dimensional array accesses, neighborhood accesses in regular and irregular graphs, and conditionals benefit from indexed SRF access. These constructs enable new classes of applications to be executed efficiently on stream processors. Our simulations of benchmarks from target application classes showed performance improvements of 1.03x to 4.1x and memory bandwidth reductions of up to 95% over a sequentially accessed SRF. In addition, since indexed SRF access provides several performance benefits beyond added locality capture, it is able to outperform cache-based implementations for certain classes of applications while avoiding the much higher area overheads of caches.

Our proposed design leverages current SRAM implementation techniques to support indexed access with 11%-22% area overhead over a sequentially accessed SRF, which translates to a modest 1.5%-3% increase in overall die area of a typical stream processor. In addition, there is no adverse impact in terms performance and power for applications that do not require indexed SRF accesses. As stream programming extends to a wider range of applications and technology scaling makes off-chip bandwidth increasingly expensive, we believe indexed SRF access will become increasingly important.

Much of the future work related to SRF indexing lies in exporting its capabilities to the application programmer.

We believe that in addition to the low-level API described in section 4.7, exporting application constructs that benefit from SRF indexing via high-level APIs within the context of sequential streaming is also an attractive approach. This allows the programmer interface to maintain the abstraction of linear streams while enabling the compilation tools to automatically identify opportunities for SRF indexing. From an architectural perspective, we are exploring support for data structures that require both reads and writes simultaneously in the SRF. Unlike streams, which are read only or write only for the duration of a kernel's execution, read-write data structures allow even more flexibility for application-specific tasks as well as system-level uses such as spilling local registers to the SRF. We also intend to evaluate the impact of sparse interconnects for the address and data networks used for cross-lane accesses.

8. Acknowledgements

The authors thank the Merrimac and Imagine groups for their expertise and software tools support. We also thank the reviewers for their valuable feedback.

9. References

[1] A. Peleg and U. Weiser. MMX Technology Extension to the Intel Architecture. In *IEEE Micro*, pp. 42-50, Aug. 1996.

[2] M. Tremblay, J.M. O'Connor, V. Narayanan, and L. He. VIS Speeds New Media Processing. In *IEEE Micro*, pp. 10-29, Aug. 1996

[3] K. Diefendorff, P.K. Dubey, R. Hochsprung, and H. Scales. AltiVec Extension to PowerPC Accelerates Media Processing. In *IEEE Micro*, pp. 85-95, Mar. 2000.

[4] R.B. Lee. Subword Parallelism with MAX-2. In *IEEE Micro*, pp. 51-59, Aug. 1996.

[5] R.M. Russell. The Cray-1 Computer System. In *Comm. of the ACM 21, 1*, pp. 63-72, Jan. 1978.

[6] J. Wawrzynek, K. Asanovic, B. Kingsbury, J. Beck, D. Johnson, and N. Morgan. Spert II: A Vector Microprocessor System. In *IEEE Computer*, pp. 79-86, Mar. 1996.

[7] C. Kozyrakis. Scalable Vector Media-processors for Embedded Systems. Ph.D. Thesis, Univ. of California at Berkeley, May 2002.

[8] A. Kunimatsu, N. Ide, el. al. Vector Unit Architecture for Emotion Synthesis. In *IEEE Micro*, pp. 40-47, Mar. 2000.

[9] R. Espasa, F. Ardanaz et. al. Tarantula: A Vector Extension to the Alpha Architecture. In *Proc. of the 29th Intl. Symposium on Computer Architecture*, pp. 281-292, May. 2002.

[10] S. Rixner, W.J. Dally, U.J. Kapasi, B. Khailany, A. Lopez-Lagunas, P. Mattson, and J.D. Owens. A Bandwidth-Efficient Architecture for Media Processing. In *Proc. of the 31st Annual Intl. Symposium on Microarchitecture*, pp. 3-13, Nov. 1998.

[11] J.D. Owens, W.J. Dally, U.J. Kapasi, S. Rixner, P. Mattson, and B. Mowery. Polygon Rendering on a Stream Architecture. In *Proc. of the 2000 SIGGRAPH/Eurographics Workshop on Graphics Hardware*, pp. 23-32, Aug. 2000.

[12] W.J. Dally, P. Hanrahan et. al. Merrimac: Supercomputing with Streams. To appear in *Supercomputing '03*, Nov. 2003.

[13] B. Khailany, W.J. Dally, A. Chang, U.J. Kapasi, J. Namkoong, and B. Towles. VLSI Design and Verification of the Imagine Processor. In *Proc. of the IEEE Intl. Conference on Computer Design*, pp. 289-296, Sep. 2002.

[14] D.B. Loveman. Program Improvement by Source-to-Source Transformation. In *Jour. of the ACM 24, 1*, pp. 121-145, Jan. 1977.

[15] J.E. Smith, G. Faanes, and R. Sugumar. Vector Instruction Set Support for Conditional Operations. In *Proc. of the 27th Annual Symposium on Computer Architecture*, pp. 260-269, Jun. 2000.

[16] U.J. Kapasi, W.J. Dally, S. Rixner, P. Mattson, J.D. Owens, and B. Khailany. Efficient Conditional Operations for Data-parallel Architectures. In *Proc. of the 33rd Annual Intl. Symposium on Microarchitecture*, pp. 159-170, Dec. 2000.

[17] B. Amrutur and M. Horowitz. Speed and Power Scaling of SRAMs. In *Journal of Solid-State Circuits*, pp. 175-185, Feb. 2000.

[18] P. Shivakumar and N.P. Jouppi. CACTI 3.0: An Integrated Cache Timing, Power, and Area Model. WRL Research Rpt., Aug. 2001.

[19] P. Mattson. A Programming System for the Imagine Media Processor. Ph.D. Thesis, Stanford University, Mar. 2002.

[20] D. Bhandarkar and R. Bunner. VAX Vector Architecture. In *Proc. of the 17th Annual Symposium on Computer Architecture*, pp. 128-138, May 1990.

[21] M. Brandt, J. Brooks, M. Cahir, T. Hewitt, E. Lopez-Pineda, and D. Sandness. The Benchmarker's Guide for CRAY SV1 Systems. Cray Inc., Jul. 2000.

[22] L.I. Kontothanassis, R.A. Sugumar, G.J. Faanes, J.E. Smith, and M.L. Scott. Cache Performance in Vector Supercomputers. In *Proc. of Supercomputing'94*, pp. 255-264, Nov. 1994.

[23] K. Asanovic. Vector Microprocessors. Ph.D. thesis, University of California at Berkeley, May 1998.

[24] J. Daemen and V. Rijmen. AES Proposal: Rijndael,1999.

[25] V. Fischer and M. Drutarovsky. Two Methods of Rijndael Implementation in Reconfigurable Hardware. In *Proc. of the Third Intl. Workshop on Cryptographic Hardware and Embedded Systems*, pp. 77-92, May 2001.

[26] W. Stallings. Cryptography and Network Security, 2nd ed. Prentice Hall, 1998.

[27] B. Khailany, W.J. Dally, S. Rixner, U.J. Kapasi, J.D. Owens, and B. Towles. Exploring the VLSI Scalability of Stream Processors. In *Proc. of the Ninth Symposium on High Performance Computer Architecture*, pp. 153-164, Feb. 2003.

[28] A. Pajuelo, A. Gonzalez, and M. Valero. Speculative Dynamic Vectorization. In *Proc. of the 29th Annual Intl. Symposium on Computer Architecture*, pp. 271-280, May 2002.

[29] The Connection Machine CM-5 Technical Summary. Thinking Machines Corp., Nov. 1992.

[30] N.P. Jouppi, J. Bertoni, and D.W. Wall. A Unified Vector/Scalar Floating-Point Architecture. WRL Research Report, July 1989.

[31] J. Corbal, R. Espasa, and M. Valero. Three-Dimensional Memory Vectorization for High Bandwidth Media Memory Systems. In *Proc. of the 35th Annual Intl. Symposium on Microarchitecture*, pp. 149-160, Nov. 2002.

[32] M.L. Simmons and H.J. Wasserman. Performance Comparison of the Cray-2 and Cray X-MP/416. In *Proc. of Supercomputing '88*, pp. 288-295, Nov. 1988.

[33] T.C. Mowry, M.S. Lam, and A. Gupta. Design and Evaluation of a Compiler Algorithm for Prefetching. In *Proc. of the Fifth Intl. Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 62-73, Oct. 1992.

[34] S. Agarwala, P. Koeppen, et. al. A 600MHz VLIW DSP. In *Digest of Technical Papers, IEEE Solid State Circuits Conference 2002*, pp. 38-39, Feb. 2002.

[35] TMS320C54x DSP Reference Set, Volume 1: CPU and Peripherals. Texas Instruments Inc., Mar. 2001.

[36] K. Mai, T. Paaske, N. Jayasena, R. Ho, W.J. Dally, and M. Horowitz. Smart Memories: A Modular Reconfigurable Architecture. In *Proc. of the 27th Annual Intl. Symposium on Computer Architecture*, pp. 161–171, Jun. 2000.

[37] K. Sankaralingam, R. Nagarajan, H. Liu, C. Kim, J. Huh, D.C. Burger, S.W. Keckler, and C.R. Moore. Exploiting ILP, TLP and DLP with the Polymorphous TRIPS Architecture, In *Proc. of the 30th Annual Intl. Symposium on Computer Architecture*, pp. 422-433, Jun. 2003.